Constantin Gălățan Susana Gălățan

Curs de C#

Programare în Visual C# 2008 Express Edition

L&S Info-mat

Tiparul executat la S.C. LuminaTipo S.R.L.
Str. Luigi Galvani nr. 20 bis, Sector 2, Bucureşti,
office@luminatipo.com

Descrierea CIP a Bibliotecii Naționale a României

GĂLĂȚAN, CONSTANTIN
Curs de C# : programare în Visual C# 2008 Express Edition / Constantin Gălățan, Susana Gălățan. - Bucureşti : Editura L & S Info-mat, 2008
ISBN 978-973-7658-16-6

I. Gălățan, Susana

004.43 C#

Editura L&S INFO-MAT:



Adresa: Str. Stânjeneilor nr. 6, bl. 30, sc. A, et. 1, apt. 11, sector 4, Bucureşti;
Tel/Fax: 031-105.62.84;
Mobil: 0722-530.390; 0722-57.37.01;

E-mail: comenzi@ls-infomat.ro;
office@ls-infomat.ro;

http://

Magazin online: www.ls-infomat.ro

Catalog online: www.manuale-de-informatica.ro

Biblioteca Digitală de Informatică "TUDOR SORIN"
www.infobits.ro

CUPRINS

Partea I

Limbajul C#

Capitolul 1

C# şi platforma .NET

Numele limbajului **C#** a fost inspirat din notaţia ♯ (*diez)* din muzică, care indică faptul că nota muzicală urmată de ♯ este mai înaltă cu un semiton. Este o similitudine cu numele limbajului C++, unde ++ reprezintă atât incrementarea unei variabile cu valoarea 1, dar şi faptul că C++ este mai mult decât limbajul C.

Limbajul C# a fost dezvoltat în cadrul Microsoft. Principalii creatori ai limbajului sunt Anders Hejlsberg, Scott Wiltamuth şi Peter Golde. Prima implementare C# larg distribuită a fost lansată de către Microsoft ca parte a iniţiativei **.NET** în iulie 2000. Din acel moment, se poate vorbi despre o evoluţie spectaculoasă. Mii de programatori de C, C++ şi Java, au migrat cu uşurinţă spre C#, graţie asemănării acestor limbaje, dar mai ales calităţilor noului limbaj. La acest moment, C# şi-a câştigat şi atrage în continuare numeroşi adepţi, devenind unul dintre cele mai utilizate limbaje din lume.

Creatorii C# au intenţionat să înzestreze limbajul cu mai multe facilităţi. Succesul de care se bucură în prezent, confirmă calităţile sale:

- Este un limbaj de programare simplu, modern, de utilitate generală, cu productivitate mare în programare.
- Este un limbaj *orientat pe obiecte*.
- Permite dezvoltarea de aplicaţii industriale robuste, durabile.
- Oferă suport complet pentru dezvoltarea de componente software, foarte necesare de pildă în medii distribuite. De altfel, se poate caracteriza C# ca fiind nu numai *orientat obiect*, ci şi *orientat spre componente*.

La aceste caracteristici generale se adaugă şi alte trăsături, cum este de pildă suportul pentru internaţionalizare, adică posibilitatea de a scrie aplicaţii care pot fi adaptate cu uşurinţă pentru a fi utilizate în diferite regiuni ale lumii unde se vorbesc limbi diferite, fără să fie necesare pentru aceasta schimbări în arhitectura software.

În strânsă legatură cu ***Arhitectura .NET*** (***.NET Framework***) pe care funcţionează, C# gestionează în mod automat memoria utilizată. Eliberarea memoriei ocupate (*garbage collection*) de către obiectele care nu mai sunt necesare aplicaţiei, este o facilitate importantă a limbajului. Programatorii nu mai trebuie să decidă singuri, aşa cum o fac de pildă în C++, care este locul şi momentul în care obiectele trebuie distruse.

În C# se scriu de asemenea aplicaţii pentru sisteme complexe care funcţionează sub o mare varietate de sisteme de operare, cât şi pentru sisteme

dedicate (*embeded systems*). Acestea din urmă se întind pe o arie largă, de la dispozitive portabile cum ar fi ceasuri digitale, telefoane mobile, MP3 playere, până la dispozitive staționare ca semafoare de trafic, sau controlere pentru automatizarea producției.

Din punct de vedere sintactic C# derivă din limbajul C++, dar include şi influențe din alte limbaje, mai ales Java.

.NET Framework

Arhitectura .NET este o componentă software care oferă un mediu de programare şi de execuție a aplicațiilor pentru sistemele de operare Microsoft. Este inclusă în sistemele de operare *Windows Server 2008* şi *Windows Vista* şi poate fi instalată pe *Windows XP* şi *Windows Server 2003*.

.NET Framework este un mediu care permite dezvoltarea şi rularea aplicațiilor şi a serviciilor Web, independente de platformă.

Limbajul C# se află într-o strânsă legatură cu arhitectura .NET. Inițial, C# a fost dezvoltat de către Microsoft pentru crearea codului platformei .Net, la fel cum destinația inițială a limbajului C a fost aceea de a implementa sistemul de operare UNIX. .NET pune la dispoziție o colecție impresionantă de clase organizate în biblioteci, pe care C# le utilizează.

Este momentul să precizăm că C# funcționează având .NET ca infrastructură, dar .NET suportă şi alte limbaje, cum este C++, Visual Basic sau Java. În oricare dintre aceste limbaje programați, aveți la dispoziție aceleaşi biblioteci de clase. .NET se realizează în acest fel **interoperabilitatea limbajelor**.

.NET este constituit din două entități importante:

- ***Common Language Runtime*** (**CLR**)
  Acesta este **mediul de execuție** al programelor. Este modulul care se ocupă cu managementul şi execuția codului scris în limbaje specifice **.NET**. **CLR** furnizează de asemenea servicii importante, cum sunt securitatea aplicațiilor, portabilitatea acestora, managementul memoriei şi tratarea excepțiilor.

- ***Base Class Library***
  Este vorba despre *Biblioteca de Clase .NET*. Această bibliotecă acoperă o arie largă a necesităților de programare, incluzând interfața cu utilizatorul, conectarea cu bazele de date şi accesarea datelor, dezvoltarea aplicațiilor web, comunicarea în rețele şi altele. Codul bibliotecii este precompilat, fiind încapsulat de regulă în funcții, numite metode, pe care programatorul le poate apela din propriul program. La rândul lor, metodele aparțin claselor, iar clasele sunt organizate şi separate între ele cu ajutorul spațiilor de nume (*namespaces*). Despre toate aceste noțiuni vom vorbi pe larg în capitolele următoare. Ceea ce trebuie reținut pentru moment, este că programatorii combină propriul cod cu codul Bibliotecii de Clase .NET pentru producerea de aplicații.

Compilarea programelor pe platforma .NET

Limbaje interpretate

Când programaţi într-un limbaj de programare interpretat, veţi scrie codul şi veţi rula aplicaţia. Cu ajutorul unui interpretor specific limbajului, fiecare linie de cod este interpretată chiar în momentul rulării şi este preschimbată imediat în cod maşină şi executată. Partea bună este că puteţi rula codul până la primul bug, puteţi înlătura eroarea în cod iar apoi să continuaţi execuţia codului. Limbajul nu necesită timp de compilare şi de legare. Partea neplăcută este că limbajele interpretate sunt lente. Limbajul Basic a fost iniţial un limbaj interpretat, dar mai apoi, începând cu anii 1980, au apărut şi versiuni compilate. Marea majoritate a limbajelor de scriptare Web sunt de asemenea limbaje interpretate.

Limbaje compilate

Codul scris într-un asemenea limbaj, numit *cod sursă*, este translatat de către compilator într-un cod apropiat de nivelul maşinii, numit cod executabil (de exemplu codul conţinut în fişierele (*.exe). Dacă în timpul compilării apar erori, atunci este necesar să le înlăturaţi, după care veţi compila din nou. Dacă aplicaţia trece acum de compilare fără erori de sintaxă, atunci se va produce codul executabil şi veţi putea să rulaţi aplicaţia. Limbajele C şi C++ sunt exemple clasice de limbaje compilate.

Din această perspectivă C# este un limbaj compilat. Dar nu în sensul descris mai sus. Ca să înţelegeţi, este necesar să ştiţi că în urma compilării unui program C#, nu se crează un cod executabil. Se creează un fişier numit ***assembly*** care de regulă se identifică cu extensia **.exe** sau **.dll**. Un asemenea fişier nu poate fi executat pe un sistem pe care nu există infrastructura **.NET**. Fişierul conţine un tip special de cod, numit ***Limbaj Intermediar***, pe scurt **CIL** (*Common Intermediate Language*). Limbajul **CIL** defineşte un set de instrucţiuni portabile, independente de orice tip de procesor şi platformă.

Figura ilustrează procesul de creare a codului executabil pe platforma .NET.

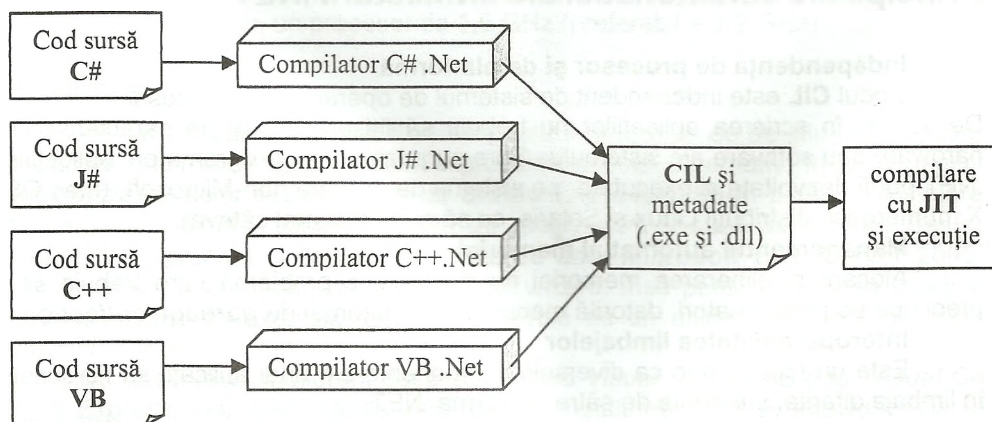


Fig. 1.1 *Procesul de compilare pe platforma .NET*

În momentul în care un program este executat, **CLR** activează un compilator special, numit **JIT** (*just in time*). Acesta preia codul **CIL** şi îl transformă în cod executabil. Transformarea se face "la cerere", în sensul că o secvenţă de cod se compilează doar în momentul în care este utilizată pentru prima oară. Un program compilat în format **CIL** poate rula pe orice sistem pe care s-a instalat ***Common Language Runtime***. Aceasta asigură portabilitatea aplicaţiilor **.NET**.

Fişierul .exe sau .dll produs la compilare conţine pe lângă codul **CIL**, aşa numitele ***metadate***. Metadatele descriu datele utilizate de către aplicaţie.

Instrumente de dezvoltare a aplicaţiilor .NET

Platforma **.NET** actuală a ajuns la versiunea **3.5**. Microsoft pune la dispoziţia programatorilor două unelte pentru dezvoltarea aplicaţiilor:

1. **Visual Studio .NET** şi varianta *free* **Visual Studio Express 2008**.
2. **.NET Framework SDK.**

Pachetul de dezvoltare a aplicaţiilor pentru **.NET 3.5** (*Microsoft .NET Framework 3.5 SDK*[1]), include:

- **.NET Framework**
- **Compilatorare** în linie de comandă pentru limbajele de programare: **C#**, **C++** , **Visual Basic**, şi **Jscript**.
- Instrumente pentru crearea, depanarea şi configurarea aplicaţiilor .NET.
- Exemple şi documentaţie.

Trebuie să ştiţi că există compilatoare pentru platforma **.NET** create de anumite firme sau organizaţii, pentru limbajele **Smaltalk**, **Perl**, **Cobol** sau **Pascal**, ca să enumerăm doar câteva disponibile pe piaţă.

Principalele caracteristici ale arhitecturii .NET

Independenţa de procesor şi de platformă

Codul **CIL** este independent de sistemul de operare şi de procesor.
De aceea, în scrierea aplicaţiilor nu trebuie să fiţi preocupaţi de caracteristicile hardware sau software ale sistemului. Spre surpriza multor programatori, aplicaţiile .NET pot fi dezvoltate şi executate pe sisteme de operare non-Microsoft, (Mac OS X, numeroase distribuţii Linux şi Solaris, ca să numim numai câteva).

Managementul automat al memoriei

Alocara şi eliberarea memoriei nu mai este o problemă care trebuie să-i preocupe pe programatori, datorită mecanismului automat de ***garbage collection***.

Interoperabilitatea limbajelor

Este un fapt comun ca diversele componente ale unei aplicaţii să fie scrise în limbaje diferite, suportate de către platforma .NET.

---

[1] ***Software Development Kit*** – Kit de Dezvoltare a Aplicaţiilor

Securitate

.NET furnizează un model comun de securitate, valabil pentru toate aplicaţiile, care include un mecanism unificat de *tratare a excepţiilor* . O excepţie este un eveniment neprevăzut, care întrerupe execuţia unui program, atunci când de pildă, se execută o intrucţiune ilegală.

Portabilitate

Un program scris pentru platforma .NET poate rula fără nici o modificare pe oricare sistem pe care platforma este instalată.

Caracteristicilor de mai sus li se adaugă şi altele, care ies însă din cadrul acestei lucrări.

Mediul Integrat Visual C# 2008 Express Edition

Visual C# 2008 Express Edition (pe scut: **VCSE**), este un mediu free de dezvoltare a aplicaţiilor produs de Microsoft. Este un **IDE** (*integrated development environment*), care oferă un set de instrumente, între care un editor de cod pentru scrierea programelor C#, compilator, depanator, instrumente pentru *build automation* (automatizarea procesului de compilare) şi altele. Kit-ul de instalare *C# Express*, include *Platforma .NET 3.5*, iar aceasta la rândul ei include între altele *Biblioteca de Clase*.

Cerinţe de sistem

Sistemele de operare suportate sunt: *Windows Server 2003*, *Windows Vista*; Windows XP.

- Pentru Microsoft Windows XP, Service Pack 2
  - minim 192 MB de RAM (preferabil cel puţin 384 MB)
  - cel puţin un procesor de 1 GHz (preferabil > 1.6 GHz)
- Pentru Microsoft Windows Vista şi Microsoft Windows Server 2003
  - minim 768 MB de RAM (preferabil cel puţin 1 GB)
  - cel puţin un procesor de 1,6 GHz (preferabil > 2.2 GHz)

Instalare

Visual C# 2008 Express Edition poate fi descărcat de pe site-ul Microsoft, la adresa http://www.microsoft.com/express/download/. Alternativ, în josul paginii aveţi opţiunea de a descărca **Visual Studio Express Edition** pentru o instalare offline. Visual Studio conţine mediile de programare **Visual C#**, **Visual Basic**, **Visual C++**, precum şi serverul de baze de date **Microsoft SQL Server 2005**. Instalarea se face simplu, cu ajutorul unui wizard, însă pentru montarea imaginii DVD-ului (fişier cu extensia ISO) aveţi nevoie de un utilitar cum ar fi de pildă *Daemon Tools*.

Ca observaţie, indiferent dacă veţi instala *Visual Studio* sau doar *Visual C# 2008 Express*, veţi opta întotdeauna pentru instalarea *.NET Framework*, *Visual C#*, *MS SQL Server* şi **MSDN** (*Micorsoft Developer Network*). MSDN conţine o foarte bogată documentaţie de care nu vă puteţi lipsi când dezvoltaţi aplicaţii.

Există şi versiunile Visual C# 2008 şi Visual Studio 2008 cu facilităţi suplimentare, dar care nu sunt gratuite.

Ajunşi în acest punct, dorim să atragem atenţia asupra faptului că o prejudecată răspândită privind dezvoltarea .NET este aceea că programatorii trebuie să instaleze Visual Studio ca să poată crea aplicaţii C#. Nu este adevărat. Puteţi să compilaţi şi să rulaţi orice tip de program .NET folosind kit-ul de dezvoltare a aplicaţiilor ***.NET Framework 3.5 Software Development Kit*** (**SDK**), care este downloadabil în mod gratuit. Acest SDK vă pune la dispoziţie compilatoare, utilitare în linie de comandă, conţine *Biblioteca de Clase .Net*, exemple de cod şi o documentaţie completă.

Se pot crea aplicaţii C# în două moduri diferite:

1. Folosind *Notepad* sau oricare editor de text şi apoi compilarea în linie de comandă.
2. Utilizând *Visual Studio Express Edition*. Aceasta este metoda preferabilă, datorită sprijinului considerabil pe care-l oferă mediul integrat în dezvoltarea aplicaţiilor, mai ales a acelora cu interfaţă cu utilizatorul.

Compilare în linie de comandă

Deşi s-ar putea ca niciodată să nu vă decideţi să dezvoltaţi aplicaţii mari folosind compilatorul C# în linie de comandă, este totuşi important să înţelegeţi cum se lucrează în linie de comandă fie doar şi pentru următoarele motive:

- Nu aveţi o copie a Visual C# 2008 Express, sau sistemul pe care-nu satisface cerinţele hardware sau software minimale:
- Doriţi să proiectaţi un ***build tool*** automatizat, aşa cum este *MSBuild* sau *Nant*, care necesită să cunoaşteţi opţiunile în linie de comandă ale utilitarelor.
- Doriţi o înţelegere profundă a C#, doriţi să vedeţi ce se petrece în spatele scenei atunci când utilizaţi IDE-uri pentru dezvoltarea aplicaţiilor.

Primul program C#

Este o tradiţie păstrată în timp, ca o carte de programare să înceapă cu programul care afişează pe ecran "`Salut lume !`". Vom compila şi executa acest program C# în linie de comandă, urmând paşii de mai jos:

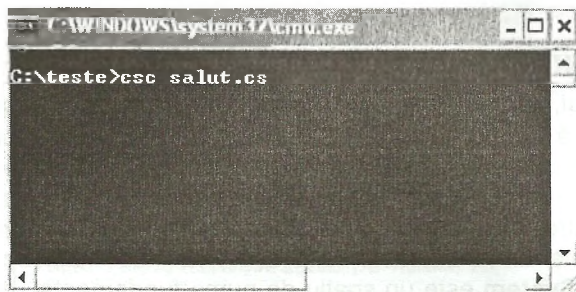
1. Editarea codului sursă

Deschideţi *Notepad* şi scrieţi următorul cod:

```
using System;
class Salut
{
    static void Main()
    {
        Console.WriteLine("Salut lume!");
    }
}
```

În C# fişierele sursă au extensia **cs**. Salvaţi fişierul cu numele **salut.cs** într-un folder oarecare, de exemplu **C:\teste**.

2. **Compilarea în linie de comandă**
   La promptul de comandă, compilaţi astfel: **csc salut.cs**

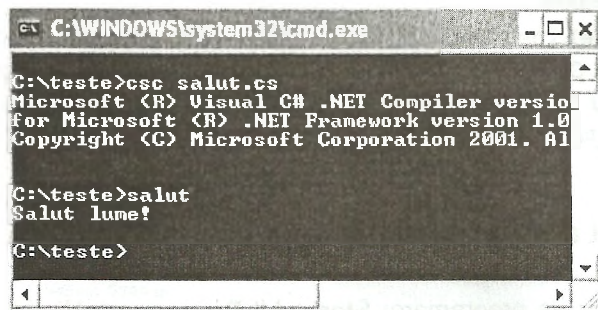


Aşadar, compilatorul **csc.exe** (*C sharp compiler*) primeşte ca argument în linie de comandă numele fişierului sursă. Veţi constata că în urma compilării, în folderul C:\teste s-a creat fişierul **salut.exe**. Acesta este un ***assembly***, conţinând codul intermediar **CIL**. Executarea lui este posibilă doar pe sistemele care au infrastructura .NET.

3. **Executarea programului**
   Rularea programului se face simplu, prin tastarea numelui fişierului **.exe**:

c:\teste>`salut`

Programul afişează:
`Salut lume!`



Setarea variabilei de mediu PATH

Dacă platforma .NET a fost instalată odată cu *Visual Studio* sau **VCSE**, atunci la instalare puteţi opta ca variabila de mediu **PATH** să reţină în mod implicit căile spre utilitarele în linie de comadă, inclusiv spre compilator, ceea ce vă scuteşte de efortul de a le introduce manual. Dacă PATH nu reţine aceste căi, atunci nu veţi putea lucra în linie de comandă, deoarece sistemul de operare nu va găsi aceste utilitare.

În cazul în care nu ați ales varianta setării PATH la instalare, o puteți face ulterior astfel:

- La promptul de comandă, mutați-vă în subdirectorul **Common7\Tools** al instalării.
- Rulați fișierul de comenzi `VSVARS32.bat` scriind: `VSVARS32`.

O scurtă analiză a programului salut.cs

Prima linie a programului, `using System;` spune implementării C# prin intermediul *directivei* ***using***, că se vor utiliza clase aflate în *spațiul de nume* **System**. Așa cum veți vedea în capitolul următor, C# a preluat din C++ *spațiile de nume* (*namespaces*). Spațiile de nume funcționează ca niște containere, în care se definesc nume de clase pentru evitarea conflictelor de nume, dar și pentru separarea logică. Toate clasele din *Biblioteca de Clase .NET* se definesc în interiorul unor spații de nume. **System** este un spațiu de nume fundamental, care definește între altele, clasa **Console**. Aceasta din urmă, conține *metode* (funcțiile din C#) care scriu și citesc date de la consolă.

Observăm că la fel ca în C++, există metoda **Main()**, care se declară întotdeauna **static**.

IMPORTANT!

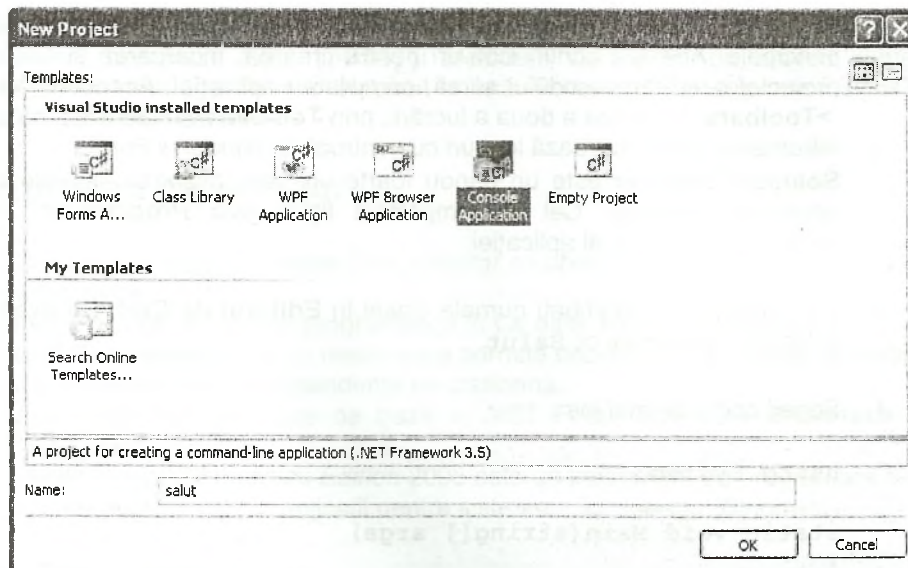
O deosebire față de C++ este faptul că trebuie să existe o clasă, în cazul de față clasa **Salut**, care găzduiește metoda **Main()**. Aceasta, deoarece C# este un ***limbaj pur orientat obiect***, și nu pot exista definiții de funcții în afara claselor și nici variabile globale.

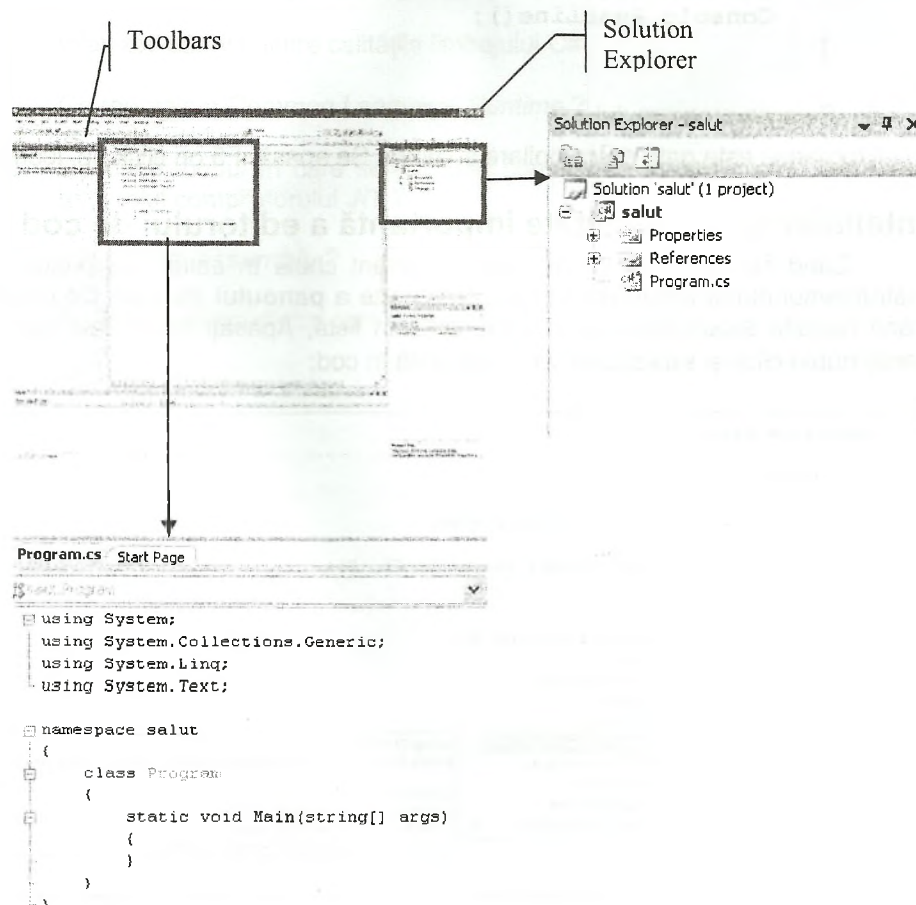
În acest sens, C# seamănă cu Java. Totuși, în Java numele clasei care conține metoda **main()** trebuie să coincidă în mod obligatoriu cu numele fișierului. Dacă veți schimba mai sus numele clasei, de exemplu **Hello**, o să constatați că nu veți avea nici o problemă la compilare.

Linia `Console.WriteLine("Salut lume!");` este apelul metodei statice `WriteLine` din clasa `Console`. O să revenim asupra metodelor statice în capitolele următoare. Pentru moment, rețineți că o metodă statică se apelează astfel: `nume_clasa.nume_metodă`.

Crearea unei aplicații de tip consolă

1. Deschideți mediul integrat de programare: **Start->All Programs-> Visual C# 2008 Express Edition**.
2. În meniul **File**, selectați ***New Project***. Se deschide dialogul ***New Project***. Acesta permite alegerea diferitor tipuri de aplicații. Selectați **Console Application** ca tip de proiect și schimbați numele aplicației în **salut**. Click **OK**.
   Visual C# Express Edition 2008 crează un nou folder pentru proiect, cu același nume cu cel al proiectului. Deschide de asemenea fereastra principală și **Editorul de Cod** unde veți intra și veți modifica codul sursă C#.


New Project
Templates:
Visual Studio installed templates
Windows Forms A...
Class Library
WPF Application
WPF Browser Application
Console Application
Empty Project
My Templates
Search Online Templates...
A project for creating a command-line application (.NET Framework 3.5)
Name:
salut
OK
Cancel


Toolbars
Solution Explorer
Solution Explorer - salut
Solution 'salut' (1 project)
salut
Properties
References
Program.cs
Program.cs
Start Page
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace salut
{
class Program
{
static void Main(string[] args)
{
}
}
}

Barele de instrumente (Toolbars) sunt în partea de sus a ferestrei principale. Acestea conțin icon-uri pentru crearea, încărcarea şi salvarea proiectelor, editarea codului sursă, compilarea aplicației. Accesare: **View->Toolbars**. În partea a doua a lucrării, prin **Toolbar** vom identifica bara de intrumente care păstrează iconuri cu controalele *Windows Forms*.

Solution Explorer este un panou foarte util care afişează fişierele care constituie proiectul. Cel mai important fişier este ***Program.cs***." care conține codul sursă al aplicației.

3. Este permis să schimbați numele clasei în **Editorul de Cod**. De exemplu schimbați `Program` cu `Salut`.

4. Scrieți codul de mai jos :

```
using System;

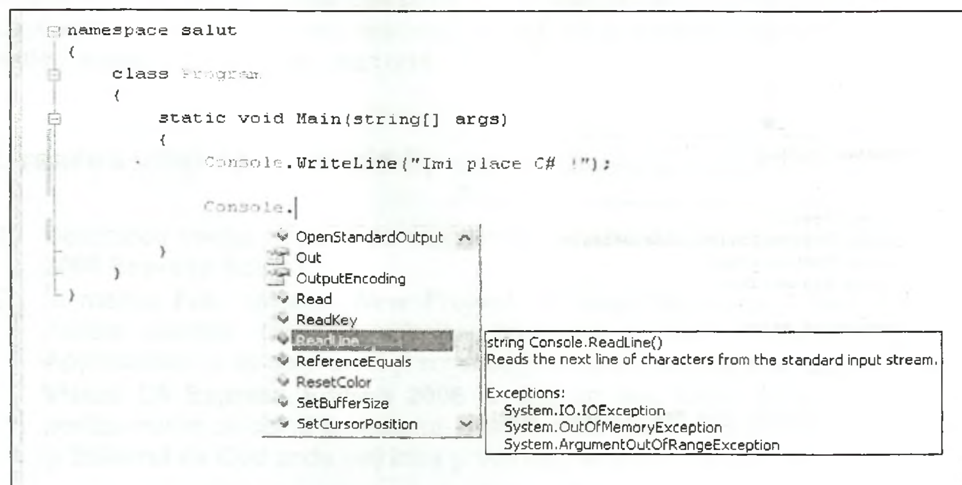
static void Main(string[] args)
{
    Console.WriteLine("Imi place C# !");
    Console.ReadLine();
}
```

5. Rularea programului.

Programul este gata de compilare şi rulare. Se apasă **F5** ori click pe iconul

IntelliSense – o facilitate importantă a editorului de cod

Când se scrie un nume sau un cuvânt cheie în editor, se poate utiliza instrumentul numit **IntelliSense** care este parte a **panoului de cod**. De exemplu, când metoda **ReadLine** apare evidențiată în listă, Apăsați Enter sau Tab sau faceți dublu-click şi **ReadLine** va fi adăugată în cod:



Avantajul utilizării **IntelliSense** este pe de o parte faptul că programatorul nu trebuie să memoreze toate tipurile şi metodele vizibile într-un anumit context deoarece acestea apar în mod automat în listă. Pe de altă parte, poate fi sigur că ceea ce scrie este corect.

Rezumatul capitolului

- C# este un limbaj din familia C++, *orientat pe obiecte*, cu mare productivitate în programare.
- Infrastructura pe care se programează în C# este **.NET Framework**.
- **.NET Framework** este un mediu care permite dezvoltarea şi rularea aplicaţiilor şi a serviciilor Web, independente de platformă.
- Una dintre componentele de bază a *.NET Framework* este ***Biblioteca de Clase .NET***.
- Mediul *Visual C# Express Edition* 2008 este un instrument de programare *free*, cu care puteţi dezvolta aplicaţii pentru sistemele de operare *Windows*.

Întrebări şi exerciţii

1. Precizaţi câteva dintre calităţile limbajului C#.

2. Ce reprezintă *Common Language Runtime* ?

3. Descrieţi modul în care se produce compilarea unui program C#. Care este rolul compilatorului *JIT* ?

4. Ce este un ***assembly*** ?

5. Descrieţi câteva dintre caracteristicile Platformei .NET.

Capitolul 2

Limbajul C# . Introducere

Limbajul C# este deosebit de simplu, cu numai 80 de cuvinte cheie şi 12 tipuri incorporate. Dar C# este în acelaşi timp un limbaj foarte productiv din punct de vedere al programării aplicaţiilor şi pe deplin adaptat conceptelor moderne de programare.

Fiind un limbaj orientat pe obiecte, o caracteristică esenţială a sa este suportul pentru definirea şi lucrul cu clasele. Clasele definesc tipuri noi de date. Variabilele de tip clasă se numesc obiecte. Uneori un obiect este abstract, cum ar fi o tabelă de date sau un *thread* (program sau proces lansat în execuţie). Alteori obiectele sunt mai tangibile, cum sunt un buton sau o fereastră a aplicaţiei. Obiectele sunt un element fundamental în programare datorită rolului lor în modelarea problemelor din practică care trebuie rezolvate. Programarea orientată pe obiecte se sprijină pe trei piloni fundamentali: **încapsularea datelor**, **moştenire** şi **polimorfism**.

Clasele C# conţin ***câmpuri*** şi ***proprietăţi*** care reţin informaţii despre obiecte şi ***metode*** (funcţii aparţinând clasei). Metodele încapsulează cod care descrie ceea ce poate face obiectul, acţiunile şi capabilităţile sale. Obiectele la rândul lor, interacţionează, comunică între ele.

C# este un limbaj care oferă suport explicit pentru tratarea **evenimentelor**. Vom reveni pe larg asupra tuturor acestor concepte în capitolele următoare.

C# admite **programarea generică**. Programarea generică este un stil de programare diferit de programarea orientată pe obiecte. Scopul programării generice este scrierea de cod care să fie independent de tipul datelor. De exemplu, imaginaţi-vă o funcţie care primeşte ca parametru un şir numeric pe care îl sortează. Algoritmul de sortare nu depinde de tipul elementelor şirului. Acestea pot fi de pildă întregi sau reale. În această situaţie în C# puteţi defini o funcţie generică, care va folosi după caz, şiruri întregi sau reale.

C#, aşa cum s-a văzut în capitolul precedent utilizează un ***garbage collector*** (colector de deşeuri) pentru eliberarea memoriei ocupate de obiectele de care aplicaţia nu mai are nevoie. Obiectele C# se alocă întotdeauna dinamic, adică în timpul execuţiei programaului şi se distrug în mod automat.

Structura unui program simplu C#

O aplicaţie de tip Consolă

Creaţi în VCSE o aplicaţie de tip consolă şi rulaţi programul următor:

```
/*
    Programul declarară variabile, initializează
    variabile şi face atribuiri
*/
```

```
namespace SpatiulMeu
{
    class Test
    {
        static void Main(string[] args)
        {
            int a;      // Declararea unei variabile locale
            a = 20;     // Atribuire
            double b = 10;    // Declarare şi iniţializare
            string s = "obiect de tip string";

            System.Console.WriteLine("a = " + a);
            System.Console.WriteLine("b = " + b);
            System.Console.WriteLine("s este " + s);
        }
    }
}
```

Cu **Ctrl + F5** compilaţi şi lansaţi programul, care va afişa pe ecran:

```
a = 20
b = 10
s este obiect de tip string
```

Vom analiza în continuare structura acestui program.

Metoda `Main` - punctul de intrare în aplicaţie

Fiecare program necesită un punct de intrare (*entry point*) şi în cazul C#, aceasta este metoda statică **`Main`**. La lansarea în execuţie a programului, mediul de execuţie .NET (CLR) caută şi apelează metoda **`Main`**. Un program poate să conţină mai multe clase, dar numai una dintre acestea defineşte metoda **`Main`**.

Tipuri abstracte

Programul declară un tip (clasa cu numele **`Test`**) şi un membru al clasei (metoda **`Main`**). Este o diferenţă faţă de C++, unde de regulă tipurile abstracte se declară în fişiere *header*, iar definiţiile acestora în fişiere .cpp separate.

Variabile locale

Metodele unei clase pot să declare variabile locale, aşa cum este variabila întreagă **`a`**, variabila reală **`b`** şi variabila de tip **`string`**, s. Tipul **`string`** este un tip definit în *Biblioteca de Clase .NET*, fiind destinat lucrului cu şiruri de caractere. Variabilele locale pot fi iniţializate cu valori la momentul declarării.

Comentarii

C# admite comentarii multilinie (`/* ... */`) şi comentarii de o singură linie (`//`), întocmai ca C++.

Clase

C# lucrează cu clase. Fiind un limbaj pur orientat obiect, nu puteţi scrie un program C# fără clase. Totul se petrece în interiorul claselor. Tot ceea ce aparţine unei clase, va fi definit în interiorul ei. Nu există variabile globale şi nici funcţii globale. Un program trebuie să aibă cel puţin o clasă, aceea care conţine metoda **Main**. O clasă se defineşte cu ajutorul cuvântului cheie **class**.

Exemplu:

```
class Persoana
{
    // membrii clasei
}
```

Observaţie:

Programatorii C++ trebuie să fie avizaţi de faptul că în C# după ultima paranteză închisă a clasei, nu se pune ; .

Spaţii de nume definite de programator

Există spaţii de nume predefinite. Ne referim la acelea care fac parte din *Biblioteca de Clase .NET*. În acelaşi timp C# permite programatorilor să-şi creeze propriile spaţii de nume pentru organizarea logică a aplicaţiilor mari. Aţi remarcat desigur **namespace SpatiulMeu** definit în primul program. Este bine de reţinut că prezenţa lui este opţională şi că atât clasa care conţine metoda **Main**, cât şi celelalte clase ale programului pot să fie incluse sau nu în interiorul unui *namespace*.

Calificarea completă a numelor cu ajutorul operatorului .

Afişările pe ecran s-au făcut prin apelul metodei statice **WriteLine** din clasa **Console**. Clasa **Console** este definită în spaţiul de nume **System**. Pentru utilizarea claselor C# din *Biblioteca de Clase .NET*, dar şi a metodelor statice ale acestor clase, există alternativele:

1. **Utilizarea directivei using;**

   Directiva **using** face vizibile în program toate numele de clase dintr-un spaţiu de nume. Reamintim că spaţiile de nume sunt containere logice care servesc la separarea numelor de tipuri într-o bibliotecă de clase sau într-un program.

   Exemple:

   - **using System;** - toate numele definite în spaţiul de nume **System** devin accesibile.
   - **using System.Windows.Forms** - clasele din spaţiul de nume **Forms** devin vizibile în program. Observaţi că spaţiile de nume se pot include unele în altele. De pildă **System** conţine mai multe clase, dar şi spaţii de nume printre care spaţiul de nume **Windows**. Spaţiul de nume **Windows** la rândului lui include mai multe spaţii de nume, între care şi **Forms**.

2. **Calificarea completă a numelui**
Calificarea completă a numelui (*fully qualified name*) presupune precizarea căii complete până la numele metodei. În programul anterior, apelul `System.Console.WriteLine("a = " + a);` ilustrează modul corect de apelare. Operatorul `.` serveşte la separarea spaţiilor de nume. Se porneşte de la spaţiul de nume exterior (`System`) şi se continuă cu specificarea tuturor spaţiilor de nume până se ajunge la metoda dorită.
Să scriem două programe simple:

```
// Primul program foloseşte  directiva using
using System;

class Test
{
     static void Main()
     {
         Console.WriteLine("C#");
     }
}

// Al doilea program precizează calea completă
// a numelui metodei
class Test
{
     static void Main()
     {
         System.Console.WriteLine("C#");
     }
}
```

Cele două programe compilează şi rulează identic.

Parametri în linia de comandă

Metoda `Main` poate avea un parametru. Acesta este un şir de stringuri, care reţine argumentele în linia de comandă care se transmit programului. Acestea sunt informaţii care pot fi pasate programului de către utilizator. Creaţi un proiect de tip consolă cu numele **arg**. În editorul de cod introduceţi programul:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
            Console.Write(args[i] + " ");
    }
}
```

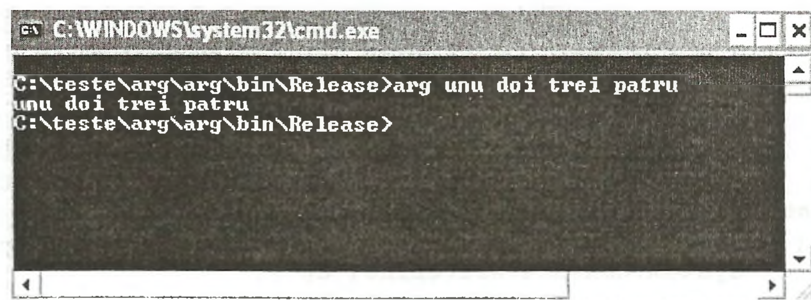
Compilați proiectul (faceți **Build**) cu **F6**. În folderul **\bin\Release** al aplicației se va produce fişierul ***assembly*** cu numele **arg.exe**.

OBSERVAȚII

- Vom reveni în capitolul următor asupra buclei `for`. În acest moment rețineți sintaxa preluată din limbajul C.
- `Length` este o proprietate publică a clasei abstracte **Array**, care returnează numărul de elemente ale şirului. Toate tablourile unidimensionale din C# moştenesc clasa `Array`, deci pot utiliza proprietățile acesteia.

Transmiterea argumentelor de la promptul sistemului de operare

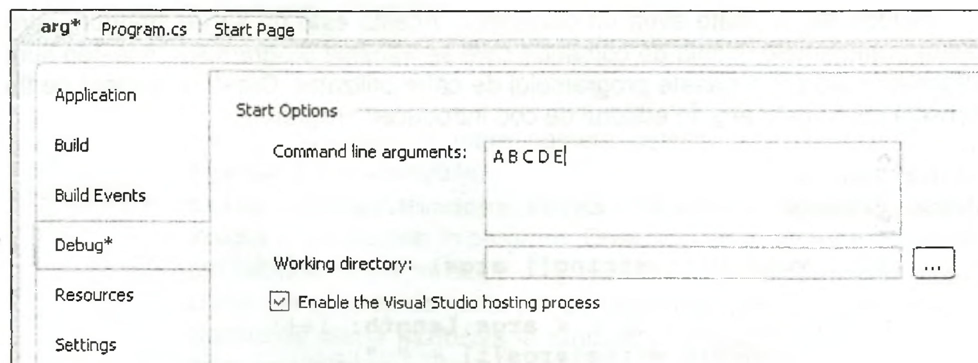
Interpretorul în linie de comandă al sistemului de operare este **cmd.exe**. Acționați **Start->Run** şi introduceți comanda **cmd**. La promptul sistemului schimbați calea în folderul **\bin\Release** al proiectului. Tastați numele programului, urmat de cateva argumente separate prin spații: `arg unu doi trei patru`. Programul afişează pe ecran: `unu doi trei patru`



Argumentele sunt de regulă informații de care programul are nevoie, de exemplu numele fişierelor de intrare şi de ieşire.

Transmiterea argumentelor în linie de comandă din Visual C# Express

În fereastra *Solution Explorer*, faceți dublu click pe iconul *Properties*. În fereastra din dreapta, click pe tab-ul *Debug*. Introduceți argumentele în linie de comandă în *TextBox*-ul cu eticheta *Command Line Arguments:*



Faceți ***Build*** şi rulați acționând **Ctrl + F5**. Programul afişează: `A B C D E`

Variante ale metodei Main

La crearea unei aplicații de tip consolă, **VCSE** generează următoarea definiție pentru metoda `Main`:

```
static void Main(string[] args) { /*...*/ }
```

Este declarată `static`, are tipul de retur `void` şi un şir de stringuri, ca şi parametru de intrare. C# admite şi alte *signaturi* ale funcției Main, ca mai jos:

```
static int Main(string[] args) { /*...*/ }  // Tip de retur int,
                                            // un şir de stringuri ca argument
static void Main() { /*...*/ }              // Fără tip de retur, fără argumente

static int Main() { /*...*/ }               // Tip de retur int, fără argumente
```

Veți alege un prototip sau altul, în funcție de necesități. Dacă doriți ca aplicația să întoarcă sistemului de operare o valoare întreagă, de pildă un cod de eroare când programul se termină, atunci veți prefera o signatură care returnează `int`. Dacă aplicația foloseşte argumente transmise de către utilizator din linie de comandă, atunci veți opta pentru o versiune cu un parametru de tip şir de stringuri.

IMPORTANT

- *C# este case-sensitive. Aceasta înseamnă că* **`Main`** *este corect, iar* **`main`** *greşit, că* **`WriteLine`** *nu este tot una cu* **`writeline`**.
- *Cuvintele cheie din C#, (`return`, `int`, `void`, `if`, etc) se scriu cu litere mici.*
- *Pentru tipurile definite de programator, pentru spațiile de nume şi pentru metode, există convenția pe care este indicat să o respectați şi anume: prima literă este mare şi fiecare cuvânt conținut începe tot cu literă mare.*
  Exemple: `Console.ReadLine()` – *metodă*
  `System.TimeZone` – *spațiu de nume*
  `System.Data.SqlClient` – *clasă*

Rolul cuvântului cheie `static` în prototipul metodei Main

`Main` este membră a unei clase, de pildă clasa `TestAfisare`. Dar o clasă nu este altceva decât o specificație a unui tip de date. În lipsa unui obiect de tip `TestAfisare`, nu există nici un membru al clasei. Totuşi, `Main` trebuie să fie apelată pentru că este punct de intrare în program. Pentru ruperea cercului vicios, se defineşte `Main` ca metodă statică. Metodele statice au tocmai această proprietate interesantă, că ele există şi pot fi apelate în lipsa unui obiect de tipul clasei.

Accesul public sau privat la metoda Main

La fel ca în cazul altor limbaje orientate obiect, în C# se specifică modul de acces la membrii clasei cu ajutorul *modificatorilor de acces* `public` sau `private`. Vom reveni în capitolul următor asupra acestor cuvinte cheie. Pe scurt, cuvântul `private` aplicat unui membru de clasă face ca acel membru să nu poată fi accesat din exteriorul clasei. *Visual C# 2008 Express* defineşte mod implicit metoda `Main` ca fiind `private`, pentru ca alte aplicaţii să nu o poată apela. Aveţi libertatea să specificaţi modificatorul public sau private. Construcţiile următoare sunt legale:

```
public static void Main()
{
}

private static void Main()
{
}
```

Rezumatul capitolului

- Metoda `Main` este punctul de intrare într-un program C#.
- Un program trebuie să aibă cel puţin o clasă, aceea care conţine metoda `Main`.
- `Main` are câteva versiuni. Prin tabloul de stringuri pe care `Main` îl are ca parametru, se pot pasa programului argumente în linia de comandă.
- C# nu admite variabile globale. Toate variabilele declarate sunt locale.
- Toate clasele .NET sunt definite în interiorul unor spaţii de nume.

Întrebări şi exerciţii

1. Ce se înţelege prin ***garbage collection*** ?
2. Care este rolul cuvântului cheie static în *signatura* metodei `Main` ?
3. Ce se înţelege prin *fully qualified name* ?
4. Scrieţi un program C# care se rulează într-o aplicaţie de tip consolă. Programul afişează toţi parametrii transmişi în linie de comandă. Nu se vor utiliza directive de compilare.
5. Ce rol are operatorul **.** într-un program C# ?

Capitolul 3

Fundamentele limbajului C#

Acest capitol descrie elemente de bază ale limbajului C#: tipurile de date, sintaxa, expresiile, operatorii, directivele de procesare.
Toate programele din acest capitol se pot testa cu ajutorul unei aplicații de tip consolă în *Visual C# 2008 Express Edition*.

Tipuri

C# este un limbaj **puternic tipizat** *(strongly-typed)*. Un asemenea limbaj impune programatorului să declare un tip pentru fiecare obiect creat, iar compilatorul verifică compatibilitatea dintre tipurile obiectelor şi valorile care le sunt atribuite.

Tipurile sistem din C#, se împart în două categorii:

- **Tipuri predefinite** *(built-in)*, cum sunt `int`, `char`, etc.
- **Tipuri definite de programator** (*user-defined*). Acestea se definesc cu ajutorul claselor, structurilor sau interfețelor.

Un alt sistem de clasificare a tipurilor în C# împarte tipurile astfel:

- **Tipuri referință**
- **Tipuri valoare**

Tipurile valoare sunt variabile locale care se memorează în segmentul de stivă al programului, iar tipurile referință se alocă dinamic în **Heap**.

C# admite şi **tipul pointer**, însă pointerii sunt folosiți foare rar şi doar în situațiile când se lucrează cu *unmanaged code*, adică cod nespecific platformei .Net.

Tipuri predefinite

C# oferă toate tipurile de bază necesare unui limbaj de programare modern. În C# toate tipurile sunt obiecte, chiar şi cele mai elementare tipuri. Cuvintele cheie pentru tipurile predefinite (`int`, `char`, `double`, etc.) sunt mapate direct peste tipuri `struct` din spațiul de nume `System`. De exemplu, cuvântului cheie `int` îi corespunde clasa `System.Int32` din *Biblioteca de Clase .NET*. Aveți libertatea să declarați o variabilă întreagă astfel:

`System.Int32 x;` , în loc de `int x;`

Maparea tipurilor primitive peste tipuri din .NET asigură faptul că tipurile create în C# pot interacționa cu tipuri create în alte limbaje care lucrează pe platforma .Net.
Tipurile referință predefinite sunt `object` şi `string`.

Tipul `object`

Tipul `object` este rădăcina întregii ierarhii de tipuri şi clasa de bază a tuturor claselor din **.NET Framework**. Aceasta înseamnă pe de o parte că toate clasele din .NET moştenesc clasa `object` şi pe de alată parte că oricare clasă definită de programator derivă în mod implicit din `object`.

Tipul `string`

Tipul `string` reţine un şir de caractere din setul **Unicode**. **Unicode** este un standard care permite reprezentarea tuturor caracterelor scrise în orice limbă scrisă, nu doar în limbile de circulaţie internaţională (în jur de 100.000 carcatere).

Tabelul 3.1 ***Tipuri C# predefinite***

Tip	Dimensiune (bytes)	Tip .Net asociat	Descriere
`object`	variabilă	`Object`	Tipul de bază al tuturor tipurilor
`string`	variabilă	`String`	Tip şir de caractere.
`byte`	1	`Byte`	Tip întreg fără semn. [0, 255]
`char`	2	`Char`	Tip caracter. Reţine carcatere Unicode
`bool`	1	`Boolean`	Valori **true** sau **false**
`sbyte`	1	`SByte`	Tip întreg cu semn. [-128, 127]
`short`	2	`int16`	Tip întreg cu semn. [-32768, 32767]
`ushort`	2	`UInt16`	Tip întreg fără semn. [0, 65535]
`int`	4	`Int32`	Tip întreg cu semn. [-2.147.483.648, 2.147.483.647]
`uint`	4	`Uint32`	Tip întreg fără semn. [0, 4.294.967.295]
`float`	4	`Single`	Tip real cu virgulă mobilă, simplă precizie. Valori cuprinse între +/-1,5 x 10^{-45} şi +/-3,4 x 10^{38} cu 7 cifre semnificative
`double`	8	`Double`	Tip real cu virgulă mobilă, dublă precizie. Valori cuprinse între +/-5,0 x 10^{-324} şi +/-1,8 x 10^{308} cu 15 cifre semnificative
`decimal`	12	`Decimal`	Tip zecimal cu virgulă fixă. Precizie de cel puţin 28 cifre semnificative.
`long`	8	`Int64`	Tip întreg cu semn. [-2^{63}, 2^{63} - 1]
`ulong`	8	`UInt64`	Tip întreg fără semn. [0, 2^{64}]

Tipul `decimal` se foloseşte în calcule financiare, atunci când erorile de rotunjire datorate virgulei mobile sunt inacceptabile.

Declarări de variabile

Toate declarările de mai jos sunt legale:

```
object o1, o2 = null;
string s = "Salut";
sbyte b = 6;
short h = 13;
```

```
ushort u = 7;
int a = 10;
long p = 3, q = 8L;
ulong x = 2UL, y = 5L, z = 10U, w = 8;
float f1 = -4.5, f2 = 3.8F;
double d1 = 1.3, d2 = 0.2D;
char c = 'T';
Decimal d = 12.4M;     // M desemnează o constantă decimal
```

Tipuri definite de programator

În C# programatorii pot defini propriile tipuri cu ajutorul cuvintelor cheie : **class**, **struct**, **interface**, **delegate** şi **enum**. Tipul tablou este de asemenea un tip *user-defined*.

Tipul class

În C# clasele se creează cu ajutorul cuvântului cheie **class**:

```
class nume_clasa
{
    modificator_acces Tip nume_membru;
    // alţi membri
}
```

Exemplu:

```
class Intreg
{
    public int x;  // x - câmp (membru) al clasei
}
```

De reţinut :

- Variabilele de tip clasă se numesc **obiecte** sau **instanţe** ale clasei.
- Crearea unui obiect de tip clasă, se numeşte **instanţiere**.

Instanţierea se face cu ajutorul operatorului **new**, urmat de apelul constructorului clasei. Vom reveni asupra acestor noţiuni în capitolul următor. Vom crea două obiecte de tip Intreg:

```
Intreg r1, r2;          // Declară două referinţe
r1 = new Intreg();      // Creează un obiect referit de r1
r2 = new Intreg();      // Creează un obiect referit de r2
```

Mai sus, **r1** şi **r2** se numesc **referinţe**.

IMPORTANT

Accesarea membrilor clasei se face cu sintaxa: **`referinţa.membru`**

Modificatorul **`public`** care prefixează declararea câmpului **`x`**, indică faptul că **`x`** poate fi accesat din afara clasei sale astfel: **`r.x`**.
Ilustrăm aceste informaţii în programul *`intreg.cs`*:

```
using System;

class Intreg
{
    public int x;
}

class TestIntreg
{
    static void Main()
    {
        Intreg r = new Intreg();
        r.x = 10;     // Atribuie valoarea 10 câmpului x
        Console.Write(r.x);    // Afisează 10
    }
}
```

Observaţi că un program C# poate să conţină una sau mai multe clase, dintre care una singură conţine metoda **`Main`**.

Structuri

Structurile se construiesc prin utilizarea cuvântului cheie **`struct`**. Au fost introduse în C# pentru a permite programatorului să definească tipuri valoare. Sunt similare claselor, dar le lipsesc unele caracteristici, cum ar fi moştenirea. Obiectele de tip **`struct`** se depozitează în stivă. Se creează şi se accesează mai rapid decât cele alocate dinamic. Sunt de preferat atunci când aveţi nevoie de obiecte mici ca dimensiuni, sau care se construiesc într-un ciclu.

Sintaxa este similară cu cea a claselor:

Exemplu

```
using System;

struct Persoana
{
   public string nume;
   public int varsta;
}
```

```
class TestStruct
{
    static void Main()
    {
        // Datele obiectului p se memorează în stivă
        Persoana p = new Persoana();
        p.nume = "Iulia";
        p.varsta = 8;
        Console.Write("Numele {0}\nVarsta {1} ani",
                            p.nume, p.varsta);
    }
}
```

La rulare, programul afişează:

```
Numele Iulia
Varsta 8 ani
```

Notă

- Metodele **`Write`** şi **`WriteLine`** din clasa **`Console`**, admit formatarea afişării în maniera funcţiei **`printf()`** din **C**. Modificatorii `{0}` şi `{1}` specifică locul şi ordinea în care se vor afişa valorile argumentelor `p.nume`, respectiv `p.varsta`.
- `\n` este o *secvenţă escape*, reprezentând caracterul (*newline*).

Tipuri valoare şi tipuri referinţă

În C# tipurile se clasifică în ***tipuri valoare*** şi ***tipuri referinţă***.

Tipurile valoare sunt toate tipurile predefinite (**`char`**, **`int`**, **`float`**, etc.), cu excepţia tipurilor **`string`** şi **`object`**. C# permite în plus definirea de tipuri valoare cu ajutorul cuvântului cheie **`struct`**. Tipurile valoare se caracterizează prin faptul că se memorează în segmentul de stivă al programului.

Tipurile referinţă, se definesc de către programator. În această categorie se includ tipurile *clasă*, tipurile *interfaţă*, tipurile *delegat* şi tipurile *array* (tablourile). Se numesc tipuri referinţă, deoarece variabilele folosite pentru manipularea lor sunt referinţe la obiecte alocate dinamic în ***Heap***. O variabilă de tip referinţă, păstrează de fapt adresa obiectului din ***Heap*** pe care îl referă, astfel că există o asemănare între referinţe şi pointerii C++.

Ca să putem lămuri pe deplin diferenţele între tipurile referinţă şi tipurile valoare, vom vedea cum se crează obiectele.

IMPORTANT

*Tipurile valoare diferă de **tipurile referinţă** prin faptul că variabilele de tip valoare conţin datele obiectului, în timp ce variabilele de tip referinţă conţin referinţe la obiecte.*

Pentru a lămuri aceasta, vom analiza programul:

```
class C      // Tip referință
{
}

struct S     // Tip valoare
{
}

class StackHeap
{
    static void Main(string[] args)
    {
        int x = 7;        // Variabilă locală.
        S s = new S(); // s - variabilă locală

        C r;             // Referința r se memorează pe Stivă.
        r = new C();     // Alocă în Heap un obiect referit de r
    }
}
```

Unde se vor memora variabilele **x**, **s**, **r** şi obiectele create cu **new** ? Priviți figura:

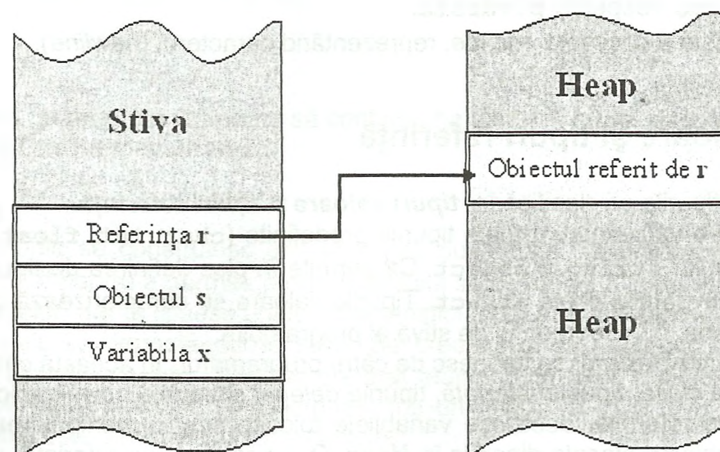


Figura 3.2. *Memorarea datelor tipurilor* ***valoare*** *şi* ***referință***

Observăm câteva aspecte importante:

- Obiectul cu numele **s** de tip **struct**, creat cu **new** se memorează în stivă.
- Referința **r** la obiectul de tip **class** se memorează în stivă, iar obiectul propriuzis creat cu **new** se alocă în **Heap**.
- Variabila **x** se memorează în stivă.

DE REȚINUT:

- Datele variabilelelor de **tip valoare**, se memorează pe stivă (Stack). O excepție de la această regulă o constituie situația în care tipuri valoare sunt incorporate unui tip referință (de pildă, sunt date membre ale unei clase). În acest caz datele se memorează în *Heap*.
- În cazul **tipurilor referință**, referințele sunt în stivă, iar datele se alocă dinamic în *Heap*.

Conversii între tipuri

Obiectele de un anume tip se pot converti în obiecte de un alt tip în mod implicit sau explicit. Tipurile predefinite suportă de asemenea conversii.

Conversiile implicite

Acest tip de conversie se produce în mod automat. O face compilatorul și aveți garanția că se produce fără pierdere de informație.

Exemplu:

```
short a = 4;
int n = a;    // Conversie implicită spre tipul int.
              // Nu se pierde informaţie
```

Conversiile explicite

Dacă încercați să atribuiți unei variabile de tip `short` valoarea unei variabile de tip `int`, compilatorul nu va efectua conversia implicită, deoarece aceasta poate cauza pierdere de informație. Aveți totuși posibilitatea de a efectua această conversie cu ajutorul operatorului de conversie explicită (*cast operator*) :

```
int n = 10000;
short s1 = n;          // Incorect. Nu va compila
short s2 = (short)n;   // Corect. Conversie explicită
                       // cu pierdere de informaţie
```

Secvența următoare ilustrează modul în care conversiile au loc în C#:

```
// Declarări
object o = null;
int a = 2000;
short b = 3;
char c = 'T';
long d = 10;
double f = 1.2;
// Testarea convertibilitatii
a = d;          // Eroare.
a = (int)d;     // Corect. Conversie explicită
o = b;          // Corect. Toate tipurile se convertesc
                //   în mod implicit la tipul object
b = o;          // Eroare.
```

```
f = b;          // Corect. Conversie implicita
b = f;          // Eroare.
ulong e = -3;   // Eroare.
a = c;          // Corect. Conversie implicita
c = a;          // Eroare.
c = (char)a;    // Corect. Conversie explicită
```

Variabile şi constante

Variabile

O variabilă este o locaţie de memorie care are un tip asociat. În programul de mai jos, **a** şi **b** sunt variabile:

```
using System;
static void Main(string[] args)
{
    int a = 2;
    int b;
    b = 3;
    System.Console.WriteLine(a + " " + b);
}
```

Programul afişează: **2 3**
Comentaţi instrucţiunea `b = 3;` Veţi constata faptul că programul are o eroare. Compilatorul anunţă că aţi încercat să utilizaţi o variabilă căreia nu i-a fost atribuită o valoare.

IMPORTANT

C# nu permite utilizarea variabilelor cărora programul nu le-a asociat anterior o valoare.

Constante

O constantă este o variabilă iniţializată, a cărei valoare nu mai poate fi modificată în program. Constantele se introduc cu ajutorul calificativului **const**:

```
using System;
static void Main(string[] args)
{
    const long x = 100;   // Corect
    const string s;      // Eroare. Constanta neinitializata
    s = "Salut";
    System.Console.WriteLine(x + " " + s);
}
```

Programul nu va compila, deoarece stringul **s** a fost declarat constant dar nu i s-a atribuit o valoare iniţială.

Enumerări

Enumerările sunt tipuri de date de **tip valoare**, create de programator în scopul de a grupa sub un nume un set de constante simbolice.

Exemplul 1:

```
using System;

enum Saptamana
{
    Luni, Marti, Miercuri, Joi,
    Vineri, Sambata, Duminica
}

class TestEnum
{
    static void Main()
    {
        Saptamana ziua = Saptamana.Luni;
        if ( ziua == Saptamana.Luni)
            Console.Write("Luni e {0} Marti e {1}",
                (int)Saptamana.Luni, (int)Saptamana.Marti);
    }
}
```

Programul afişează: `Luni e 0 Marti e 1`

De reținut:

- Membrii unei enumerări se numesc enumeratori. Lista membrilor enumerării, se separă prin virgulă.
- Variabila `ziua` de tip `enum`, reține doar unul dintre membrii enumerării (`Luni`).
- Valorile membrilor enumerării se obțin prin conversii explicite la `int` şi sunt egale cu 0, 1, 2, ... , dacă nu se specifică alte valori pentru enumeratori.

Exemplul 2

Membrilor unei enumerări li se pot asocia valori de tip întreg:

```
using System;

enum Note
{
    Mate = 10,
    Info = 9,
    Romana = 8
}
```

```
class TestEnum
{
    static void Main()
    {
        Console.Write("mate: {0}\ninfo: {1}\nromana: {2}",
            (int)Note.Mate, (int)Note.Info, (int)Note.Romana);
    }
}
```

Programul afişează:

```
mate: 10
info: 9
romana: 8
```

Enumerările sunt o alternativă puternică în raport cu constantele. Utilizarea enumerărilor face codul mai clar, mai bine documentat.

Expresii

Expresiile în C# sunt similare cu expresiile din C++ şi Java. Expresiile se construiesc folosind operanzi şi operatori şi în general întorc o valoare în urma evaluării lor. Operanzii sunt variabile de memorie, nume de metode, de tablouri, de obiecte, etc.

Tabelul de mai jos prezintă operatorii C# în ordinea descreşterii **precedenţei** (priorităţii) lor.

Tabelul 3.3. *Operatorii C#*

Grupul de operatori	Operator	Expresii	Descriere
Primari	.	x.m	Acces la membrii clasei
	()	f(x)	Apel de metodă
	[]	a[x]	Acces la elem. tablourilor
	++	x++	Postincrementare
	--	x--	Postdecrementare
	new	new T(x)	Creare de obiecte
		new T[x]	Creare de tablouri
	typeof	typeof(T)	Informaţii despre tipul **T**
	checked	checked(x)	Evalează expresia **x** în medii *checked* şi *unchecked*
	unchecked	unchecked(x)	
Unari	+ -	+x -x	Identitate şi negaţie
	!	!x	Negaţie logică
	~	~x	Negaţie pe biţi
	++	++x	Preincrementare
	--	--x	Predecrementare
	()	(T)x	Conversie explicită
Multiplicativi	* / %	x*y x/y x%y	Înmulţire, împărţire, modulo

Aditivi	`+ -`	`x+y x-y`	Adunare, scădere
Shiftare	`<<` `>>`	`x << y` `x >> y`	Deplasare pe biţi la stânga şi la dreapta
Relaţionali şi de testare a tipului	`< > <= >=`	`x<y x>y x<=y x>=y`	Operatori aritmetici
	`is`	`x is T`	`true`, dacă `x` este convertibil la `T`. Altfel, `false`
	`as`	`x as T`	Returnează `x` convertit la `T` sau `null` dacă conversia e imposibilă
Egalitate	`== !=`	`x == y x!=y`	Egal şi diferit
AND logic	`&`	`x & y`	AND logic pe biţi
XOR logic	`^`	`x ^ y`	XOR logic pe biţi
OR logic	`|`	`x | y`	OR logic pe biţi
AND condiţional	`&&`	`x && y`	Evaluează `y` numai dacă `x` este `true`
OR condiţional	`||`	`x || y`	Evaluează `y` numai dacă `x` este `false`
Condiţional	`?:`	`x ? y : z`	Evaluează `y` dacă `x` este `true`. Altfel, evaluează `z`
Atribuire	`= *= /= %= += -= <<= >>= &= ^= |=`	`x = y x %= y etc.`	Atribuire simplă şi atribuiri compuse

Observaţie:

Din tabel se constată că apelurile metodelor sunt expresii, accesarea unui element într-un tablou este o expresie, crearea de obiecte sau de tablouri sunt de asemenea expresii.

Când o expresie conţine mai mulţi operatori, precedenţa operatorilor determină ordinea de evaluare a operaţiilor.
De exemplu, `x - y / z` se evaluează ca `x - (y / z)` deoarece precedenţa operatorului / este mai mare decât cea a operatorului - .

Când un operand se găseşte între doi operatori cu aceeaşi precedenţă, atunci *asociativitatea operatorilor* controlează ordinea operaţiilor.

În afara operatorilor de atribuire, toţi operatorii binari se asociază de la stânga spre dreapta. De exemplu: `a + b + c` se asociază `(a + b) + c`.

Operatorii de atribuire şi operatorul condiţional (`?:`) se asociază de la dreapta spre stânga. De exemplu: `a = b = c` se evaluaează în ordinea `a = (b = c)`.

Precedenţa şi asociativitatea se pot controla cu ajutorul parantezelor. De exemplu: `a+b/c` împarte `b` la `c` şi apoi adună `a` la rezultat, dar `(a+b)/c` adună `a` cu `b` şi împarte rezulatul la `c`. (**[1]**)

Tablouri

Un tablou este o structură de date conţinând variabile de acelaşi tip. În C# tablourile sunt **tipuri referinţă**. Tablourile pot fi **unidimensionale** sau **multidimensionale**. Cele multidimensionale, la rândul lor pot fi dreptunghiulare sau **neregulate** (*jagged*).

Tablouri unidimensionale

Declararea tablourilor unidimensionale se face precizând un tip de dată urmat de operatorul de indexare şi numele tabloului. Crearea tabloului se face cu `new` urmat de tip şi dimesiunea tabloului cuprinsă între paranteze pătrate.

Declarare: `Tip[] a;`
Creare: `a = new Tip[n];`

Tablourile se indexează de la `0`. Un tablou cu `n` elemente este indexat de la `0` la `n-1`.

Exemplul 1:

Programul de mai jos defineşte un tablou cu **5** valori de tip `int`.

```
using System;
class TabUnidim1
{
    static void Main()
    {
        // Tablou cu 5 elemente de tip int
        int[] a;            // Declarare
        a = new int[5];     // Creare
        for (int i = 0; i < a.Length; i++)
            a[i] = i + 1;

        for (int i = 0; i < a.Length; i++)
            Console.Write("a[{0}] = {1}\n", i, a[i]);
    }
}
```

Programul afişează:

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

IMPORTANT !

Tablourile sunt în realitate **obiecte** *care moştenesc clasa* **Array**. *De aceea, metodele şi proprietăţile acestei clase pot fi folosite pentru tablouri. Proprietatea* **Length** *a clasei* **Array** *returnează numărul de elemente ale tabloului.*

Exemplul 2:

Mai jos, **a**, **b**, **c**, sunt tablouri de **double**, **char**, **string**, respectiv obiecte de tip **A**.

```
class TabUnidim2
{
    class A
    {
    }

    static void Main()
    {
        double[] a = new double[100];
        char[]   b = new char[50];
        string[] s = new string[1000];

        // Tablou de 7 obiecte de tip A
        A[] ob = new A[7];
        for (int i = 0; i < ob.Length; i++)
            if ( ob[i] == null ) //
                Console.Write("null ");

        Decimal[] d = new Decimal[10];
        for (int i = 0; i < d.Length; i++)
            Console.Write(d[i] + " ");
    }
}
```

Programul afişează:

```
null null null null null null null 0 0 0 0 0 0 0 0 0 0
```

Observaţii:

1. C# vă permite să definiţi clase imbricate (*nested*), aşa cum este cazul clasei **A**, definită în interiorul clasei **TabUndim2**.
2. Valorile implicite ale tablourilor numerice sunt setate la **0**, iar elementele referinţă sunt setate la **null**.

Iniţialializare

Tablourile unidimensionale se pot iniţializa cu valori de acelaşi tip cu tipul tabloului:

Exemple:

```
// Tablou cu 4 elemente.
int[] a = new int[] { 7, 4, 3, 0 };

// Sintaxă alternativă. Tablou cu 3 elemente
int[]   b = {10, 2, 17};

float[] c = { 1.2F, 7.9F, -8F };

string[] s1 = new string[] { "UNU", "DOI", "TREI" };

// Sintaxă alternativă. Tablou cu 3 elemente
string[] s2 = { "UNU", "DOI", "TREI" };
```

Tablouri multidimensionale

Un tablou poate avea una, două sau mai multe dimensiuni.

Exemple:

```
// Tablou bidimensional cu 3 linii şi 5 coloane
double[,] a = new double[3, 5];

// Tablou tridimensional cu dimensiunile 3, 7 şi 5
double[, ,] a = new double[3, 7, 5];
```

Accesarea elementelor

Accesarea elementelor unui tablou multidimensional se face cu sintaxa cunoscută din limbajul Pascal. De exemplu, elementul aflat pe linia 2 şi coloana 0 într-o matrice se accesează astfel: `a[2, 0]`.

Inițializare

Tablourile multidimensionale pot fi inițializate ca în exemplele următoare:

```
// Tablou cu 2 linii şi 3 coloane
int[,] x = new int[, ]{ { 1, 2, 3 }, { 2, 3, 0 } };

// Identic, dar cu sintaxa alternativa
int[,] y = { { 1, 2, 3 }, { 2, 3, 0 } };
```

DE REȚINUT

Dacă ați creat mai întâi referința, iar tabloul va fi creat ulterior, atunci sintaxa alternativă de inițializare nu se mai poate folosi. În acest caz este obligatorie utilizarea operatorului **new**.

Exemplu:

```
using System;

class TabBidim
{
    static void Main()
    {
        int[,] y;
        // y = { { 1, 2, 3 }, { 2, 3, 0 } };  // Ilegal

        int[,] x;
        x = new int[,] { { 1, 2, 3 }, { 2, 3, 0 } }; // OK

        for (int i = 0; i < 2; i++)
        {
            for (int j = 0; j < 3; j++)
                Console.Write(x[i, j] + " ");
            Console.WriteLine();
        }
    }
}
```

Ieşire:

```
1 2 3
2 3 0
```

Tablouri neregulate

Un tablou neregulat (***jagged array***) este un tablou ale cărui elemente sunt tablouri. Elementele pot fi tablouri unidimensionale sau chiar multidimensionale.

Exemplu:

Un *jagged array* cu două elemente, fiecare dintre acestea fiind un şir de `int`, se declară astfel:

```
int[][] a = new int[2][];
```

Tabloul nu este încă complet definit. Este nevoie de inițializarea fiecărui element:

```
a[0] = new int[3]; // tablou unidimensional cu 3 elemente
a[1] = new int[5]; // tablou unidimensional cu 5 elemente
```

Inițializare

C# permite câteva forme de inițializare a tablourilor neregulate:

1. Fie tabloul neregulat:

```
int[][] a = new int[3][];
```

Elementele tabloului se pot iniţializa cu sintaxa:

```
a[0] = new int[] { 2, 7, 3, 8 };
a[1] = new int[] { 0, -1 };
a[2] = new int[] { 1, 9, 4 };
```

Tabloul are întradevăr, formă neregulată.

2	7	3	8
0	-1		
1	9	4	

2. A doua variantă de iniţializare a unui tablou neregulat este următoarea:

```
int[][] a = new int[][]
{
    new int[] { 1, 4 },
    new int[] { 5, 9, 0, 7, 2 },
    new int[] { 6, 8, 3 }
};
```

3. A treia variantă de iniţializare este:

```
int[][] a =
{
    new int[] { 1, 4 },
    new int[] { 5, 9, 0, 7, 2 },
    new int[] { 6, 8, 3 }
};
```

4. Cu ajutorul cuvântului cheie `var`, se pot iniţializa atât variabile simple cât şi tablouri cu **tipuri implicite**:

```
var d = 3.5;       // var înlocuieste double
var s = "Salut";   // var înlocuieste string

// Tablou unidimensional. var inlocuieste string[]
var p = new[] { "C#", null, "C++", "Java" };
// Tablou neregulat. var inlocuieste int[][]
var q = new[]
{   new[] { 7, 4 },
    new[] { 2, 0, -4, 8 }
};
```

În fiecare caz compilatorul deduce tipul de dată din tipul constantelor de iniţializare.

Accesarea elementelor

Pentru un tablou neregulat ale cărui elemente sunt tablouri unidimensionale, accesarea se face la fel ca la tablourile din C++: `a[i][j]`.

Exemplu:

```
using System;

class JaggedArray
{
    static void Main()
    {
        int[][] a =
        {
            new int[] { 1, 4 },
            new int[] { 5, 9, 0, 7, 2 },
            new int[] { 6, 8, 3 }
        };

        for (int i = 0; i < a.Length; i++)
        {
            for (int j = 0; j < a[i].Length; j++)
                Console.Write(a[i][j] + " ");
            Console.WriteLine();
        }
    }
}
```

Ieşire:

```
1 4
5 9 0 7 2
6 8 3
```

Tablouri neregulate cu elemente de tip tablou multidimensional

Elementele unui *jagged array* pot fi tablouri multidimensionale.

Exemplu:

Elementele tabloului neregulat **a** sunt tablouri dreptunghiulare de valori **int**:

```
int[][,] a = new int[3][,]
{
    new int[,] { {2, 9}, {4, 7}, {5, 0} },
    new int[,] { {1, 5}, {3, 12} },
    new int[,] { {8, 6}, {2, 9}, {0, 6}, {3, -1} }
};
```

Accesarea elementelor tabloului se face astfel: `a[i][j, k]`, unde `i` este al i-lea tablou din `a`, iar `j` şi `k` este poziţia valorii căutate în matricea `i`. De exemplu, valoarea `-1` din `a`, se obţine: `a[2][3, 1]`.

Instrucţiuni

În C# o declarare (*statement*) poate fi: o expresie, o instrucţiune simplă, o instrucţiune compusă, o instrucţiune de control al fluxului de execuţie (**`if`**, **`for`**, **`while`**, etc.), un apel de funcţie, un bloc de cod, dar şi o declarare de variabilă locală.

Un program C# este o secvenţă de asemenea *statements* care se evaluează în ordine. În continuare, vom folosi cuvântul *instrucţiune* pentru statement. Fiecare intrucţiune se termină cu `;`. Intrucţiunile C# sunt preluate cu unele mici modificări din C şi C++. În continuare vom prezenta intrucţiunile care controlează fluxul de execuţie al programului.

Instrucţiunea de selecţie if ... else

Instrucţiunea **`if...else`** este împrumutată din limbajul C. Controlează fluxul programului prin evaluarea unei expresii booleane. În funcţie de valoarea de adevăr, selectează pentru executare o secvenţă de instrucţiuni sau o alta.

```
if (expresie_booleană)
     instrucţiunea1;
else
     instrucţiunea2;
```

sau

```
if (expresie_booleană)
{
     bloc_instrucţiuni1;
}
else
{
     bloc_instrucţiuni2;
}
```

Exemplu:

```
string s = Console.ReadLine();

int x = int.Parse(s);
if ( x  >= 0 )
     Console.Write("pozitiv");
else
     Console.Write("negativ");
```

În cazul instrucţiunilor `if` imbricate, trebuie să ţineţi seama de regula conform căreia un **`else`** se asociază cu cel mai apropiat `if` care nu are **`else`**:

```
if (expresie1)
    if (expresie2)
        instrucţiune1;
    else
        intrucţiune2; // se asociază cu if (expresie2)
```

De reţinut:

1. Metoda **`ReadLine()`** din clasa **`System.Console`** citeşte linia următoare din ***stream**-ul standard* de intrare (tastatură).
2. Metoda **`Parse(s)`**, membră a structurii **`System.Int32`** converteşte stringul **`s`** într-un număr într-un înteg pe 32 de biţi. Amintiţi-vă că tipul **`int`** este un *alias* pentru structura **`Int32`**.

Din păcate biblioteca **.NET** nu furnizează operatorul >>, aşa cum o face C++, capabil să extragă *bytes* din *stream*-uri şi să-i convertească direct spre tipul dorit (Exemplu C++: **`int x; cin >> x`**). Metodele **`Read()`** sau **`ReadLine()`** care se definesc ca membre ale diferitor clase din **.NET**, citesc întotdeauna din *stream*-urile de intrare un caracter, un bloc de caractere sau o linie şi returnează caracterul citit sau un string. Drept urmare, de câte ori aveţi nevoie să citiţi valori numerice, veţi obţine un string, pe care îl veţi converti cu **`Parse()`**. Toate tipurile de date întregi **`Int16`**, **`Int32`**, **`SByte`**, etc, definesc această metodă.

Instrucţiunea de selecţie `switch`

Instrucţiunile `if` imbricate sunt greu de scris corect, greu de citit şi greu de depanat. **`switch`** este o intrucţiune de control mai potrivită atunci când aveţi un set complex de alegeri de făcut. Sintaxa este identică cu cea din C++.

Sintaxa:

```
switch ( n )
{
    case val1: bloc_instrucţiuni1;
               break;
    case val2: bloc_instrucţiuni;
               break;
    // alte cazuri
    default: bloc_instrucţiuni;          // opţional
}
```

`n` este o variabilă de tip întreg, de tip **`char`** sau de tip **`string`**. **`val1`**, **`val2`**, ... etc, sunt valorile pe care le poate lua **`n`**. Controlul este transferat etichetei **`case`** care potriveşte valorii lui **`n`**.

Diferenţa faţă de **`switch`** din C++ este aceea că C# nu suportă căderea implicită la secţiunea următoare (*fall through*). Exemplul următor nu va compila:

```
int k = 0;
switch ( k )
{
    case 0:
    Console.WriteLine( "cazul 0" );
    // goto case 1;
    case 1:
    case 2:
    Console.WriteLine( "cazul 1 si 2" );
    break;
}
```

Căderea prin ramuri se poate face numai pentru cazuri vide, cum e `case 1`. Dacă doriți un salt direct la o anume etichetă, folosiți instrucțiunea de salt necondiționat `goto`. Dacă scoateți de sub comentariu `goto case 1;` în secvența de program precedentă, atunci pe ecran se va afişa:

```
cazul 0
cazul 1 si 2
```

Ciclul for

Este o instrucțiune identică cu cea din C++.

Sintaxa:

```
for (expresie1; expresie2; expresie3)
{
    bloc_instrucțiuni;
}
```

Dacă `bloc_instrucțiuni` e format din o singură instrucțiune, atunci acoladele sunt opționale. Se evaluează mai întâi **expresie1**. Apoi se evaluează **expresie2**. Aceasta controlează bucla. Dacă **expresie2** este evaluată **true**, se execută instrucțiunile cuprinse între acolade, apoi **expresie3**. Se reia evaluarea **expresie2** şi ciclul continuă până când **expresie2** este evaluată **false**. Apoi controlul este cedat instrucțiunii următoare ciclului `for`.

Toate expresiile sunt opționale. Dacă **expresie2** lipseşte, condiția de test este evaluată **true**:

```
for( ;; )
   Console.Write("Ciclu infinit!");
```

Ciclul foreach

foreach este nou în familia de limbaje C. Este folosit pentru iterarea printre elementele unui tablou sau ale unei colecții. Colecțiile sunt containere de tip generic, care pot reține secvențe omogene de obiecte, aşa cum vom vedea mai târziu.

Sintaxa:

```
foreach (Tip identificator in expresie)
    bloc_instrucţiuni;
```

Exemplu:

```
using System;

class TestForeach
{
    static void Main()
    {
        int[] a = {10, 20, 30};
        foreach (int x in a)
            Console.Write(x + " ");
        Console.WriteLine();

        string[] st = { "UNU", "DOI", "TREI" };
        foreach (string s in st)
            Console.Write(s + " ");
    }
}
```

Ieşire:

```
10 20 30
UNU DOI TREI
```

Ciclul while

Este împrumutat fără modificări din C++.

Sintaxa:

```
while (expresie)
{
    bloc_instrucţiuni;
}
```

Dacă **bloc_instrucţiuni** e format dintr-o o singură instrucţiune, atunci acoladele sunt opţionale. Se evaluează **expresie**. Dacă este evaluată **true**, atunci se execută **bloc_instrucţiuni**. Se reia evaluarea expresiei. Ciclul continuă cât timp este evaluată **true**.

Exemplu:

```
int x = 5;
while ( x > 0 )
   x--;
```

Ciclul do while

Este preluat fără modificări din C++. Este un ciclu cu test final, deci **bloc_instrucțiuni** se evaluează cel puțin o dată.

Sintaxă:

```
do
{
   bloc_instrucţiuni;
} while (expresie);
```

Dacă **bloc_instrucțiuni** e format dintr-o o singură instrucțiune, atunci acoladele sunt opționale. Se evaluează **bloc_instrucțiuni**, după care **expresie**. Cât timp aceasta se evaluează **true**, ciclul continuă. În momentul în care **expresie** se evaluează **false**, se transferă controlul intrucțiunii imediat următoare ciclului.

Exemplu:

```
int x = 5;
do
{
   x--;
} while ( x > 0);
```

Observație

Rezultatul evaluării expresiilor care controlează buclele C# trebuie să fie o valoare *booleană*, adică **true** sau **false**. Secvența următoare nu va compila:

```
int x = 5;
do
{
   x--;
} while ( x );   // Ilegal! x este evaluat la o valoare int
```

Instrucțiuni de salt necondiționat

C# admite instrucțiunile de salt **goto**, **break** și **continue**.

Instrucțiunea goto

Utilizarea **goto** este nerecomandabilă, deoarece produce cod greu de înțeles (*cod spaghetti*). În cazul în care doriți totuşi să o folosiți urmați paşii:

1. Creați o etichetă (identificator urmat de :)
2. **goto** etichetă

Exemplu:

```
using System;

public class GoTo
{
    static void Main()
    {
        int x = 1;
        repeta:            // Eticheta
        Console.Write( x + " ");
        x++;
        if (x <= 5)
            goto repeta; // Salt la eticheta
    }
}
```

Ieşire:

```
1 2 3 4 5
```

Instrucţiunile break şi continue

Dacă în blocul de instrucţiuni al unei bucle se execută **continue**, atunci are loc un salt la începutul buclei. Instrucţiunea **break** duce la ieşirea forţată din ciclu.

Exemplu:

```
using System;

public class TestBreakContinue
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i % 2 == 1)    // Pentru valori impare
                continue;      // salt la expresia i < 10
            Console.Write(i + " ");
            if (i == 10) break;        // "Rupe" bucla
        }
    }
}
```

Ieşire:

```
0 2 4 6 8 10
```

Sugestie
Nu este indicat să utilizați intens aceste instrucțiuni, deoarece creează puncte multiple de ieşire din cicluri, ceea ce duce la cod dificil de înțeles şi de întreținut (cod spaghetti).

Spații de nume

În capitolele **2** şi **3** s-au discutat primele noțiuni referitoare la ***spații de nume*** (*namespaces*). Am arătat că sunt containere logice care găzduiesc definiții ale tipurilor de date. Au fost introduse în C# pentru evitarea conflictelor de nume în utilizarea bibliotecilor de clase, dar şi pentru organizarea şi sistematizarea tipurilor de date. De exemplu, dacă într-o aplicație utilizați două biblioteci neprotejate prin spații de nume, care provin de la furnizori diferiți, este posibil ca un acelaşi nume de clasă, să-i spunem *Button*, să fie definit în ambele biblioteci, ceea ce ar crea o coliziune în program, inacceptabilă de către compilator.

Întreaga ***Biblioteca de Clase .NET*** se găseşte în interiorul acestor *namespaces*. În afara spațiilor de nume ale bibliotecii *.NET Framework* sau ale bibliotecilor altor furnizori, C# vă permite să vă creați propriile spații de nume. Se definesc foarte simplu cu ajutorul cuvântului cheie **namespace**.
Exemplu:

```
namespace MySpace
{
    static void Main()
    {
    }
}
```

Spații de nume imbricate (*nested*)

Într-un spațiu de nume puteți defini nu numai tipuri de date ci şi alte spații de nume.
Exemplu:

```
namespace X
{
    namespace Y
    {
    }
    namespace Z
    {
        namespace W
        {
        }
    }
    static void Main()
    {
    }
}
```

În interiorul lui **X** s-au definit alte două spații de nume **Y** şi **Z** şi metoda **Main()**. În **Z** s-a definit **W**.

Accesarea numelor de tipuri

Accesarea numelor definite într-un *namespace* se face precizând toate spațiile de nume până la numele clasei dorite, separate între ele cu operatorul punct (.).

Exemplu:

Cum vom accesa numele clasei **Button** ?

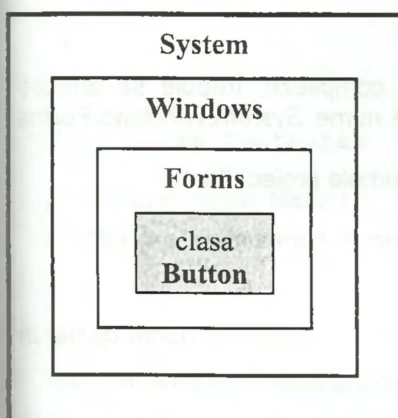


Figura 3.1 *Accesarea numelor*

În figură s-a ilustrat faptul că spațiul de nume **System** include spațiul de nume **Windows**, care la rândul lui include **Forms**. Clasa **Button**, ca de altfel toate clasele de tip fereastră se definesc în **Forms**.

Cum accesăm numele **Button** ? Vom crea un obiect de tip **Button** astfel:

```
System.Windows.Forms.Button b =
new System.Windows.Forms.Button();
```

Cam complicat, nu-i aşa ? Din fericire, există directiva **using**.

Programele C# încep de regulă cu o secțiune de directive **using**. ([4]) **using** precizează spațiile de nume cel mai utilizate în program. În felul acesta programatorul nu mai este nevoit să scrie întreaga cale (*fully qualified name*) până la clasa căutată.

a) Accesarea unui nume de clasă folosind calificarea completă a numelui acesteia:

```
public class NestedNamespaces
{
    static void Main()
    {
        System.Windows.Forms.Button b =
            new System.Windows.Forms.Button();
    }
}
```

b) Accesarea unui nume de clasă cu utilizarea directivei **using**:

```
using System.Windows.Forms;

public class NestedNamespaces
{
    static void Main()
    {
        Button b = new Button();
    }
}
```

Notă:

Pentru ca programul de la punctul **b)** să compileze, trebuie să aduceți proiectului de tip consolă o *referință*, la spațiul de nume *System.Windows.Forms* astfel:

- În ***Solution Explorer***, click dreapta pe numele proiectului.
- Alegeți *Add reference*.
- În *tab-sheet* –ul *.Net* alegeți *System.Windows.Forms* şi apăsați OK.

ATENȚIE !

O directivă **using** *nu vă permite accesul la nici un spațiu de nume definit în spațiul de nume pe care îl specificați.*

Exemplu:

```
using System;

public class WrongAcces
{
    static void Main()
    {
        Windows.Forms.Button b = new Windows.Forms.Button();
    }
}
```

Programul de mai sus nu compilează, deoarece **Windows** este un spațiu de nume imbricat în **System** şi nu puteți avea acces la acesta, ci doar la numele de tipuri (clase sau structuri) din **System**. Cu directiva de mai sus, este corect să scrieți astfel:

```
System.Windows.Forms.Button b =
                        new System.Windows.Forms.Button();
```

Alias-uri

Pentru simplificarea scrierii puteți declara nume alternative pentru spațiile de nume, astfel:

a) Introducerea unui ***alias*** pentru spaţii imbricate, definite de programator:

```
using N = N1.N2.N3;   // Creează un alias N

namespace N1
{
    namespace N2
    {
        namespace N3
        {
            public class C { }
        }
    }
}

public class TestAlias
{
    static void Main()
    {
        N.C ob = new N.C();
    }
}
```

b). Introducerea unui ***alias*** pentru spaţii de nume predefinite:

```
using F = System.Windows.Forms;

public class TestAlias
{
    static void Main()
    {
        F.ComboBox cb = new F.ComboBox();
    }
}
```

Directive de preprocesare

Înainte ca programul să fie compilat, un alt program numit *preprocesor*, rulează şi pregăteşte programul pentru compilare. Preprocesorul detectează directivele de preprocesare şi stabileşte nişte reguli de compilare în raport cu acestea. Directivele încep cu caracterul #. Le veţi introduce în situaţii în care doriţi ca unele porţiuni de cod să nu fie compilate în anumite condiţii. De exemplu, în etapa de ***build final*** a unei aplicaţii, nu doriţi să mai compilaţi porţiuni de cod pe care l-aţi folosit pentru depanare.

Directiva #define

Numele simbolice se definesc cu directiva **#define**, înaintea oricărei secvențe de cod din program. Alte directive pot fi definite şi în alte zone ale programului.

Exemplu:

```
#define DEPANARE
// Cod neafectat de directiva #define
#if DEPANARE
// Cod care se compilează dacă este definit numele DEPANARE
#else
// Cod care se compileaza daca nu este definit DEPANARE
#endif
// Cod neafectat de preprocesor
```

Când preprocesorul rulează, el caută directiva #define. Dacă aceasta există, ia notă de identificatorul **DEPANARE**. Directiva **#if** testează existența identificatorului În funcție de aceasta, setează pentru compilare o porțiune de cod sau alta.

Orice cod care nu este cuprins între **#if** şi **#endif** este compilat de către compilator.

Directiva #undefine

Dacă într-un punct în program doriți ca un anume identificator definit cu **#define** să nu mai fie definit, deci să nu mai fie cunoscut de către preprocesor, atunci folosiți directiva **#undef**:

```
#define DEPANARE
#define MAXIM

// Cod neafectat de directiva #define
#if DEPANARE
// Cod care se compilează
#endif

#undef DEPANARE

#if DEPANARE
// Cod care NU se compilează
#elif MAXIM
// Cod care compilează pentru că nu e definit DEPANARE,
// dar este definit MAXIM
#endif
```

Să observăm că **#elif** testează existența unui nume simbolic doar în cazul în care testul **#if** eşuează.

Rezumatul capitolului

- Tipurile predefinite din C# (`int`, `char`, ..., `string`), sunt mapate direct peste tipuri `struct` din spaţiul de nume `System`. De exemplu, tipului `int` îi corespunde clasa `System.Int32` din *Biblioteca de Clase .NET*.
- Tipurile definite cu ajutorul cuvintelor cheie `class` sau `struct` se numesc tipuri definite de programator.
- Variabilele de tip clasă se numesc **obiecte** sau **instanţe** ale clasei.
- ***Tipurile valoare*** sunt toate tipurile predefinite (`char`, `int`, `float`, etc.), cu excepţia tipurilor `string` şi `object`.
- ***Tipurile referinţă***, se definesc de către programator. În această categorie se includ tipurile *clasă*, tipurile *interfaţă*, tipurile *delegat* şi tipurile *array* (tablourile).
- Datele variabilelor de **tip valoare**, se memorează pe stivă (*Stack*).
- În cazul **tipurilor referinţă**, referinţele sunt în stivă, iar datele se alocă dinamic în *Heap*.
- În C# tablourile sunt **tipuri referinţă**. Tablourile pot fi **unidimensionale** sau **multidimensionale**. Cele multidimensionale, la rândul lor pot fi dreptunghiulare sau **neregulate** (*jagged*).
- Instrucţiunile de control al fluxului de execuţie a programului (`if`, `for`, `while`, etc) sunt similare sintactic cu cele din limbajul **C++**.
- Întreaga ***Biblioteca de Clase .NET*** se găseşte în interiorul spaţiilor de nume. În afara spaţiilor denume predefinite, programatorii pot să definească propriile spaţii de nume cu ajutorul cuvântului cheie `namespace`.
- Directivele de preprocesare se introduc în situaţiile în care se doreşte ca unele porţiuni de cod să nu fie compilate în anumite condiţii.

Întrebări şi exerciţii

1. Care este diferenţa dintre tipurile ***valoare*** şi tipurile ***referinţă*** ?
2. În ce condiţii se produc erori la compilare atunci când declaraţi o variabilă? Dar o constantă ?
3. În care zone ale memoriei RAM se memorează membrii unei clase ?
4. Declaraţi şi iniţializaţi un tablou bidimensional dreptunghiular cu **4** linii şi **3** coloane.
5. Declaraţi şi iniţializaţi un tablou bidimensional neregulat cu **3** linii.
6. Definiţi două spaţii de nume: `X` şi `Y`, unde `Y` este conţinut în `X`. În `Y` definiţi clasa `A`. Scrieţi două programe C# în care se instanţiază clasa `A`, utilizând moduri diferite de accesare a numelui clasei.

Capitolul 4

Programare Orientată pe Obiecte în C#

Programarea Orientată pe Obiecte (**OOP**) este cea mai influentă paradigmă de programare a timpurilor noastre. Prin "paradigmă" se înțelege stil de programare. În raport cu stilul de programare procedural, OOP reprezintă o cu totul altă abordare a programării.

În limbajulele C sau Pascal ca şi în toate celelalte limbaje procedurale, se pot defini tipuri de date şi în mod separat funcții care acționează asupra acestora. Separarea tipurilor de date de funcțiile corespunzătoare, impune programatorului eforturi considerabile pentru a menține logica grupării acestora, mai ales când aplicația are mii de linii de cod.

În OOP funcțiile sunt grupate în acelaşi obiect, împreună cu datele asupra cărora operează. Datele sunt mai complexe în OOP şi mai bine organizate. Organizarea superioară se dovedeşte foarte utilă atunci când se scriu aplicații foarte mari, când se lucrează în echipă şi când se scrie cod pentru biblioteci care vor fi folosite de clienți.

Obiecte şi clase

Limbajul C# este un limbaj *orientat-obiect*. Limbajele orientate pe obiecte au calitatea că permit programatorului să-şi creeze noi tipuri complexe de date. Tipurile se obțin prin definirea de clase. Clasele sunt baza programării orientată pe obiecte. O clasă reuneşte cod şi date care descriu o anumită categorie de obiecte. Clasa este apoi folosită pentru construirea de obiecte. Crearea unui obiect se numeşte ***instanțiere***. Obiectul creat se numeşte ***instanță*** a clasei.

Diferența dintre un obiect şi o clasă este aceeaşi ca diferența între o pisică şi descrierea unei pisici. Descrierea este clasa, iar pisica este obiectul. De altfel, clasele şi obiectele au apărut în programare tocmai din necesitatea de a modela obiectele din realitate, dar şi obiecte abstracte.

O clasă `Pisica`, specifică trăsăturile care ne interesează la o pisică: nume, rasă, greutate, culoarea blănii, etc. De asemenea, clasa descrie acțiunile pe care o pisică le poate face: să sară, să zgârie, să toarcă, etc. Pisica *Kitty*, rasa *Birmaneză*, greutatea - *4 Kg*, culoarea blănii – *gri*, este o instanță a clasei, deci un obiect de tip `Pisica`.

Marele avantaj al claselor este că încapsulează caracteristicile şi capabilitățile obiectului într-o singură unitate de cod. O clasă descrie complet obiectele de tipul său, nemaifiind nevoie în acest scop de cod exterior clasei.

Mai trebuie să rețineți următoarele: programarea orientată pe obiecte nu este doar programare cu obiecte. Între obiecte pot exista relații, obiectele pot comunica între ele, obiectele pot folosi alte obiecte.

Mecanismele fundamentale ale OOP

Toate limbajele orientate pe obiecte au câteva caracteristici comune:

1. Abstractizarea

Abstractizarea este simplificarea realităţii prin modelarea unor clase care păstrează doar trăsăturile obiectului real care sunt esenţiale şi suficiente pentru atingerea scopului propus. Membrii clasei (câmpuri, metode, etc.) definesc aceste trăsături.

2. Încapsularea

Un obiect încapsulează date şi cod care acţionează asupra acestor date. Clasa oferă protecţie membrilor săi. Nu toţi membrii clasei pot fi accesaţi din afara ei. O regulă importantă în OOP este aceea că datele membre trebuie să rămână private, să fie inaccesibile din lumea exterioară clasei, pentru a preîntâmpina modificările accidentale. Partea accesibilă a clasei o constituie membrii publici, de regulă metodele. Aceştia pot fi utilizaţi de către porţiuni din program care lucrează cu obiectele clasei. Membrii publici ai unei clase formează **interfaţa** clasei.

3. Moştenirea

Moştenirea este mecanismul prin care un limbaj orientat-obiect vă permite să definiţi noi clase care incorporează şi extind o clasă deja definită. Noile clase se numesc ***clase derivate***. Ele moştenesc atributele şi comportamentul clasei pe care o extind, fiind versiuni specializate ale clasei de bază. Clasa pe care o extindeţi se numeşte clasă de bază. Ca exemplu, să presupunem că doriţi pentru aplicaţia dumneavoastră un buton care să aibă funcţionalitatea unui buton de apăsare, dar să aibă în plus anumite proprietăţi: să fie rotund, să aibă un mic punct luminos central care clipeşte când este apăsat, etc. Aveţi două opţiuni: să scrieţi o clasă care să îndeplinească toate cerinţele sau să scrieţi o clasă **`MyButton`**, care să moştenească de pildă clasa **`System.Windows.Forms.Button`**. A doua variantă este avantajoasă, deoarece clasa **`MyButon`** moşteneşte întregul comportament şi caracteristicile unui buton obişnuit şi veţi avea nevoie să adăugaţi doar codul cerut de cerinţele suplimentare. **Reutilizarea codului** este unul dintre beneficiile importante pe care le aduce moştenirea.

4. Polimorfismul

În programarea orientată pe obiecte, *polimorfismul* (cuvânt care vine din greacă şi care înseamnă "*a avea mai multe forme*") este caracteristica unei interfeţe de a fi folosită cu acţiuni multiple. O funcţie sau un obiect au un comportament polimorfic, dacă acestea au mai mult de o singură formă.

Polimorfismul, caracterizat de expresia "*interfaţă comună, acţiuni multiple*", este deci capacitatea de a folosi în funcţie de context, mai multe forme ale unui tip sau metodă, fără să ne intereseze detaliile de implementare.

Există câteva tipuri de polimorfism. Unul dintre acestea se obţine în OOP cu ajutorul moştenirii. Să luăm un exemplu din viaţa reală: Un dresor de circ are trei animale : un şarpe, un porumbel, şi un tigru. Când vrea să cheme unul dintre ele,

le trimite un unic mesaj : "*la masa !*". Ca răspuns la acest mesaj, şarpele *se târăşte*, porumbelul *zboară*, iar tigrul *aleargă*. Să ne imaginăm clasele **`Sarpe`**, **`Porumbel`** şi **`Tigru`**. Toate clasele moştenesc clasa **`Animal`**. Fiecare clasă şi-a definit aceeaşi interfață, metoda "*LaMasa*", dar cu acțiuni diferite. Obiecte diferite (animalele) răspund în mod diferit aceluiaşi mesaj (metoda ***`LaMasa()`***). Pentru dresor nu este important cum procedează fiecare animal pentru a veni (detaliile de implementare ale metodei *LaMasa()*), ci contează doar faptul că animalul vine la masă.

Clasele C#

În afara celor cincisprezece tipuri *predefinite* în C#, puteți crea propriile tipuri de date (*user-defined types*). Există şase categorii de tipuri pe care le puteți crea:

1. Tipuri **class**
2. Tipuri **struct**
3. Tipuri **tablou**
4. Tipuri **enum**
5. Tipuri **delegate**
6. Tipuri **interface**

Odată ce ați definit un tip de dată îl puteți utiliza ca şi când ar fi un tip predefinit.

Dintre tipurile enumerate mai sus, **`class`** este cel mai important. O clasă este o structură de date care poate depozita **date** şi executa **cod**. Primele noțiuni despre clasele C# au fost discutate în capitolul **3**.

Definiție
*Clasele sunt entități logice care modelează obiecte din lumea reală sau obiecte abstracte. Clasele încapsulează **date** şi **funcții** care operează cu aceste date.*

Caracterisiticile obiectului se memorează în **câmpurile** clasei.
Capabilitățile, acțiunile obiectului, se codifică în **metodele** clasei.

- **Câmpurile** (acestea sunt ***date membre***) sunt **variabile** care rețin informații despre obiect.
- **Metodele** sunt **funcții membre** ale clasei.

Câmpurile şi metodele unei clase se numesc **membrii clasei**. O clasă poate avea un număr oricât mare de date membre şi de funcții membre. Membrii pot fi oricare combinație a următoarelor tipuri de membri:

Date membre	Funcții membre
▪ **Câmpuri**	▪ **Metode**
▪ **Constante**	▪ **Proprietăți**
	▪ **Constructori**

	▪ **Destructori** ▪ **Operatori** ▪ **Indexatori** ▪ **Evenimente**

Definirea claselor

O clasă se declară cu ajutorul cuvântului cheie `class`.

Sintaxa minimală:

```
class nume_clasă
{
      // Câmpuri
      [modificator de acces] Tip nume;

      // Metode
      [modificator de acces] TipRetur NumeMetodă(Parametri);
}
```

Modificatorii de acces controlează regiunile programului care pot accesa membrii clasei. Tabelul descrie cele şase tipuri de modificatori:

Atributul	Descriere
`public`	Acces nelimitat
`private`	Acces limitat la propria clasă
`protected`	Acces limitat la propria clasă şi la clasele derivate
`internal`	Acces limitat la programul care conţine clasa
`protected internal`	Acces limitat la program şi la clasele derivate

Dacă modificatorul de acces lipseşte, atunci membrul respectiv este în mod implicit `private`. Parantezele pătrate indică opţionalitatea.

Cum se lucrează cu obiecte ?

1. Mai întâi se defineşte clasa. **Exemplu:**

```
class X
{
    // membrii clasei
}
```

2. Acum puteţi utiliza tipul `X` pentru a crea obiecte:

```
X r = new X();
```

`r` este referinţă la obiectul de tip `X` care se alocă dinamic cu `new X()` în (*Heap*-ul sistemului). `X()` este constructorul clasei `X`.

Despre constructori vom discuta într-unul dintre paragrafele următoare.

3. Folosiți referința pentru a accesa membrii clasei:

```
r.membru; // Cu operatorul . se accesează membrii clasei
```

Exemplul 1:

```
class Program
{
      static void Main()
      {
      }
}
```

Clasa `Program` are ca membru doar metoda statică `Main()`. Această metodă este punctul de intrare în program (*entry point*). Membrii statici ai unei clase există independent de obiectele clasei. Din acest motiv, `Main()` poate fi apelată de către platforma .NET, fără să fie nevoie de un obiect de tip `Program`.

Exemplul 2:

```
class Test
{
    int x = 100;               // Câmp privat

    static void Main()
    {
//      x = 10;                // Gresit! x trebuie să aparţină
                               //         unui obiect
        Test r;                // Declară o referinţă
        r = new Test();        // Crează un obiect
        r.x = 20;              // Corect. x este privat, dar
    }                          // Main() e membră a clasei
}
```

Exemplul 3:

```
using System;

class Test
{
    private int x;             // Câmp privat

     // Metode publice
     public int GetX()
     {
        return x;
     }
```

```
    public void SetX(int n)
    {
        x = n;              // Metodele clasei au acces
    }                       // la câmpurile private
}

class Program
{
    static void Main()
    {
        Test r;             // Declară o referință
        r = new Test();     // Creează un obiect
//      r.x = 20;           // Greşit. x este private
        r.SetX(10);         // Corect. SetX() este public
        Console.WriteLine(r.GetX());
    }
}
```

Ieşire: `10`

Programul de mai sus are două clase. În clasa **`Program`**, metoda **`Main()`**, crează un obiect de tip **`Test`**, care mai apoi este utilizat pentru accesarea membrilor săi publici.

Metode. Parametrii metodelor.

Metoda este un bloc de cod care poate fi apelat din diferite părţi ale programului şi chiar din alte programe. Când o metodă este apelată, îşi execută codul, apoi transferă controlul codului care a apelat-o. Metodele pot fi implementate ca membri ai claselor sau ai structurilor sau ca prototipuri în interfeţe. Metodele pot să aibă sau să nu aibă parametri şi pot să returneze sau nu valori. Dacă nu returnează valori, returnează **`void`**. De asemenea, metodele pot fi ***statice*** sau ***ne-statice***.

ATENŢIE !

Spre deosebire de C sau C++, în C# nu există metode globale. Metodele nu se pot declara în afara unui tip de date.

Sintaxa minimală a unei metode este:

```
[ModificatorAcces] TipRetur NumeMetodă(ListaParametri)
{
    // codul metodei
}
```

Apelul unei metode:

```
NumeMetodă(ListaArgumente);
```

Argumentele (sau *parametrii actuali*) sunt valorile sau obiectele cu care se apelează metodele.

Parametrii unei metode pot fi precedaţi în C# de câteva cuvinte cheie, numite *modificatori*. Modificatorii sunt: `ref`, `out`, `params`. Aceştia controlează modul în care argumentele sunt transmise unei metode.

Parametri *valoare*

În mod implicit, *tipurile valoare* sunt pasate metodelor ***prin valoare***. Aceasta înseamnă că atunci când un obiect de tip valoare este transmis ca argument unei metode, o copie temporară a obiectului este creată de către metodă pe stivă. Când metoda îşi termină execuţia, cópiile temporare sunt distruse. În general aveţi nevoie doar de valoarea argumentelor, însă sunt situaţii când doriţi să modificaţi valoarea argumentelor. În acest caz, mecanismul transmiterii prin valoare nu vă ajută.

Exemplu:

```
using System;

class S
{
    public void Schimba(int x, int y) // x, y - parametri
    {                                 //       formali
        int aux;          // x, y, aux - variabile locale
        aux = x; x = y; y = aux;
    }
}

class Test
{
    static void Main()
    {
        S s = new S();
        int a = 10, b = 20; // a, b - variabile locale

        // a, b - argumente pasate prin valoare
        s.Schimba(a, b);
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

Programul afişează: `a = 10, b = 20`

Observăm că **a** şi **b** au rămas nemodificaţi în urma apelului, deoarece la transmiterea prin valoare **x** şi **y** sunt cópii locale ale argumentelor **a** şi **b**. Argumentele nu trebuie să fie în mod obligatoriu variabile. Poate fi argument orice expresie care se evaluează la tipul parametrului formal.

Parametri *referinţă*

Dorim să modificăm programul anterior, astfel încât valorile **a** şi **b** să se modifice după apel. Se aplică modificatorul **ref** asupra parametrilor formali şi asupra parametrilor actuali astfel:

```
public void Schimba(ref int x, ref int y)
{
    int aux;
    aux = x; x = y; y = aux;
}

int a = 10, b = 20;            // Inițializare obligatorie
s.Schimba(ref a, ref b);       // Apel
//s.Schimba(2*a, b - 1); // Greșit! Argumentele nu
                         // sunt variabile de memorie
```

Rulați programul. Va afişa: `a = 20, b = 10`

Când dorim să pasăm argumente prin referință, atunci:

- Trebuie utilizat modificatorul `ref` atât în definiție, cât şi în apelul metodei.
- Este obligatoriu ca argumentul transmis funcției să fie o variabilă, iar acestei variabile trebuie să-i fie atribuită o valoare înainte de a fi utilizată ca parametru actual.

Parametrii referință au următoarele **caracteristici**:

- Nu alocă memorie pe stiva programului pentru parametrii formali.
- Parametrul formal este de fapt un alt nume (un *alias*) pentru variabila de apel, referind aceeaşi zonă de memorie. Din această cauză, orice modificare asupra valorii parametrului formal în timpul execuției metodei, se exercită direct asupra parametrului actual şi persistă după apelul metodei.

Parametri de ieşire

Parametri de ieşire sunt utilizați pentru a transmite date din interiorul metodei spre codul care a apelat-o. Nu contează valorile inițiale ale argumentelor, ci doar cele finale.

Cerințe:

- Trebuie utilizat modificatorul `out` atât în definiție, cât şi în apelul metodei.
- Este obligatoriu ca argumentul transmis funcției trebuie să fie o variabilă şi nu o expresie. Acestei variabile nu este nevoie să-i fie atribuită o valoare înainte de a fi utilizată ca parametru actual.

Exemplu:

```
class OutParam
{
    public void Metoda(out int x)
    {
      // int y = x; // Eroare!
         x = 100;   // Inițializare înainte de
         x++;       // a se citi din variabila x
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        OutParam op = new OutParam();
        int z;     // Nu e nevoie de initializare
        op.Metoda(out z);
        System.Console.Write(z);
    }
}
```

La fel ca în cazul parametrilor referință, parametrii de ieşire (*output parameters*) sunt *alias*-uri pentru parametrii actuali. Orice schimbare asupra unui parametru de ieşire, se produce de fapt asupra argumentului funcției.

Parametrii de ieşire au următoarele **caracteristici**:

- În interiorul metodei, parametrilor de ieşire trebuie să li se atribuie valori înainte de a se utiliza valoarea acestora. Din acest motiv, valorile inițiale ale parametrilor actuali (argumente) sunt nenecesare şi nici nu este nevoie să inițializați argumentele înainte de apelul metodei.
- Tuturor parametrilor de ieşire trebuie să le fie atribuie valori înainte de revenirea din apel.

Parametri tablou

Parametrii tablou (*parameter array*) admit zero sau mai mulți parametri actuali pentru un acelaşi parametru formal.

Restricții:

- Poate exista doar un singur *parametru tablou* în lista de parametri formali.
- Dacă există un *parametru tablou*, atunci acesta este ultimul în lista de parametri formali.
- Pretinde modificatorul **params** în declarație, dar nu este permis în apelul metodei.

Parametrii tablou se declară încadrând tipul parametrului la stânga cu modificatorul **params**, iar la dreapta cu operatorul de indexare: `[]`.

Exemplu:

```
class Params
{
    public void Print(params int[] x)
    {
        foreach (int i in x)
            System.Console.Write(i + " ");
        System.Console.WriteLine();
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Params p = new Params();
        p.Print(1, 3);                    // (*)
        p.Print(2, 4, 6, 8);              // (*)
        int[] a = { 10, 20, 30 };
        p.Print(a);                       // (**)
    }
}
```

Ieşire:

```
1 3
2 4 6 8
10 20 30
```

Invocarea metodei se poate face în două moduri: o lista separată prin virgulă cu valori de tipul parametrului (*), sau un tablou de elemente (**).

O ultimă observaţie: în C# tablourile sunt tipuri referinţă, astfel încât parametrii tablou sunt şi ei referinţe.

Supraîncărcarea metodelor

C# suportă supraîncărcarea metodelor clasei (***overloading***). Este vorba depre procedeul prin care definiţi două sau mai multe metode cu acelaşi nume, în aceeaşi clasă. Supraîncărcarea are şi restricţii: metodele trebuie să difere prin numărul şi/sau tipul parametrilor formali.

Exemplu:

În clasa `Overloading` dorim să supraâncărcăm metoda: `void F(int a) {}`

```
class Overloading
{
    void F(int a) { }
    void F(char b) { }          // Corect
    void F(int a, char b) { } // Corect
    void F(int a, int b) { } // Corect

 // int F(int a){}              //Gresit! Tipul parametrului
}                               // formal este acelaşi
```

Membrii statici ai unei clase

În mod implicit, metodele şi câmpurile clasei sunt ne-statice. Pentru a putea invoca o metodă ne-statică, aveţi nevoie de un obiect de tipul clasei. Metodele statice se pot apela în lipsa obiectelor. Ele au comportamentul unor funcţii globale, care însă nu sunt vizibile decât în cadrul clasei. De asemenea, câmpurile declarate **static**, se alocă în memorie şi pot fi folosite, chiar dacă nu aţi construit nici un obiect.

Accesarea membrilor statici.

Fie clasa:

```
class C
{
    public static int x;
    public static void F() {}
}
```

Accesarea acestor membri se face prin numele clasei, astfel: `C.x` şi `C.F();`

Exemplul 1:

```
class Program
{
    static int x = 10;
    static void F()
    {
    }
    int y = 20;          // Membri ne-statici
    void G()
    {
    }
    static void Main(string[] args)
    {
        // Membrii statici aparţin clasei
        x = 100;         // Corect
        F();             // Corect

        Program.x = 100; // Corect! Accesare prin
        Program.F();     // numele clasei.

        // Membrii ne-statici aparţin obiectelor
     // y = 200;         // Greşit
     // G();             // Greşit
        Program p = new Program();
        p.y = 100;       // Corect
        p.G();           // Corect
    }
}
```

Observație:

C# admite inițializarea datelor membre, la momentul declarării lor, aşa ca mai sus: `int y = 20;` sau `int x = 10;`

Exemplul 2:

Când accesați membrii statici din afara clasei trebuie să folosiți sintaxa: `NumeClasă.MembruStatic;`

```
class ClasaMea
{
    public static string s = "Ionel";
    public static void F() { }
}

class Test
{
    static void Main(string[] args)
    {
    //  s = "Marcel";              // Greşit
        ClasaMea.s = "Marcel";  // Corect
    //  F();              // Greşit
        ClasaMea.F();     // Corect
    }
}
```

Constante

Dacă doriți să păstrați valori care nu se vor schimba pe parcursul execuției programului şi în acest scop vreți să le protejați la scriere, veți declara constante. În C# , constantele apar în două forme:

1. **Constante locale**.
2. **Constante membre** ale clasei.

Atât constantele membre, cât şi cele locale, se introduc cu cuvântul cheie **const**. **Constantele locale** se declară în corpul metodelor şi sunt vizibile doar în corpul acestora:

```
void Metoda(int a)
{
    const int x = 100;      // Inițializare obligatorie
    // x++             // Eroare!
    a = x;             // Corect
}
```

Constantele membre se comportă la fel ca valorile statice. Sunt vizibile în fiecare instanță (obiect) al clasei şi sunt disponibile chiar în lipsa obiectelor clasei.

Exemplu:

```
using System;

class C
{
    public const int H = 24;
}

class TestConst
{
    static void Main(string[] args)
    {
        Console.Write(C.H);
    }
}
```

De reținut:

- Constantele trebuie inițializate la declarare.
- Constantele membre se accesează la fel ca membrii statici : *NumeClasă.NumeConstantă*.

Constructori

Constructorul de instanță este o **metodă specială** a clasei, care are rolul de a creea, a construi obiecte. O clasă are cel puțin un constructor. De fiecare dată când instanțiați un obiect de tip **struct** sau **class**, se apelează constructorul clasei.

Exemplu:

```
class Masina
{
    public string marca;
    public int pret;
}

class Test
{
    static void Main(string[] args)
    {
        Masina m = new Masina(); // Apelul constructorului
    }
}
```

Observați că instanțierea obiectului presupune de fapt apelul constructorului clasei: **Masina()**. Toate clasele au un constructor. Puteți defini proprii constructori. În cazul în care nu o faceți, compilatorul va genera un constructor în mod implicit. Clasa **Masina** nu defineşte un constructor. În această situație, compilatorul generează un **constructor implicit**, fără parametri, **Masina()**, care construieşte obiectul, dar nu intreprinde altă acțiune.

Constructorul implicit inițializează câmpurile clasei cu următoarele valori implicite:

- Tipurile numerice (`int`, `short`, etc.) şi tipul `enum` cu valoarea 0.
- Tipul `bool` cu valoarea `false`.
- Tipul `char` cu `'\0'`.
- Tipurile referință (inclusiv `string` sau tablouri) cu `null`.

În mod frecvent veți dori să stabiliți starea inițială a obiectului prin setarea unor valori inițiale ale datelor membre. Pentru aceasta, veți defini unul sau mai mulți constructori. De exemplu, pentru clasa `Masina`, puteți scrie:

```
using System;

class Masina
{
    public string marca;
    public int pret;
    public Masina(string m, int p) // Constructor
    {
        marca = m;
        pret = p;
    }
}

class Test
{
    static void Main(string[] args)
    {
     // Masina m = new Masina(); // Gresit! Nu mai există
                                // constructorul implicit!
        // Se invocă constructorul Masina(string, int)
        Masina m = new Masina("Audi", 50000);
        Console.Write(m.marca + " " + m.pret);
    }
}
```

Ieşire: `Audi 50000`

ATENȚIE !

În momentul în care ați definit cel puțin un constructor, compilatorul nu mai sintetizează constructorul implicit. De aceea, dacă doriți şi unul fără parametri, trebuie să-l definiți.

Exemplu:

```
class Masina
{
    public string marca;
    public int pret;
```

```
    public Masina()                  // 1
    {
        marca = "Fiat";
        pret = 1000;
    }
    public Masina(string m)          // 2
    {
        marca = m;
        pret = 20000;
    }
    public Masina(int p)             // 3
    {
        marca = "Ford";
        pret = p;
    }
    public Masina(string m, int p) // 4
    {
        marca = m;
        pret = p;
    }
}
```

Acum puteți crea obiecte de tip **Maşina** în mai multe moduri:

```
Masina m1 = new Masina();          // constructor 1
Masina m2 = new Masina("Opel");  // constructor 2
Masina m3 = new Masina(2000);     // constructor 3
Masina m4 = new Masina("Renault", 15000);  // constructor 4
```

Caracteristicile metodelor constructor

- Sunt metode care se apelează ori de câte ori, un obiect este creat.
- Au acelaşi nume ca cel al clasei.
- Nu au tip de retur, deci nu returnează nimic, nici măcar **void**.
- Pot fi supraîncărcați, la fel ca oricare metodă a clasei.
- Pot să fie declarați **static**, pentru inițializarea membrilor statici ai clasei.
- De regulă se declară **public**, însă dacă doriți ca un anumit constructor să nu poată fi invocat din afara clasei, îl declarați **private**. Procedați în acest fel, mai ales când o clasă conține numai membri statici. În această situație este inutil să permiteți crearea de instanțe ale clasei (obiecte).

Constructori de copiere

Dacă doriți ca un obiect să fie creat ca o copie fidelă a unui alt obiect, atunci este convenabil să definiți un constructor de copiere (*copy constructor*). C# nu furnizează un constructor de copiere implicit, aşa cum oferă un constructor de instanță implicit. De aceea va trebui să-l definiți dumneavoastră.

Exemplu:

```
using System;

class C
{
    private int x;  // Câmp privat

    public C(int a) // Constructor de instanţă
    {
        x = a;
    }
    public C(C ob)  // Constructor de copiere
    {
        x = ob.x;
    }
    public int GetX() { return x; } // Metodă accesor
}

class TestCopyCtor
{
    static void Main(string[] args)
    {
        C c1 = new C(10);  // Apeleaza C(int)
        C c2 = new C(c1);  // Apeleaza C(C)
        Console.WriteLine(c2.GetX());   // Afişează: 10
    }
}
```

Din exemplu vedem că parametrul constructorului de copiere are tipul clasei, iar argumentul de apel este un obiect (`c1`) al clasei.

Cuvântul cheie `this`

`this` este o referinţă la obiectul curent. Toate metodele nestatice ale clasei posedă această referinţă şi au acces la ea.
Exemplificăm utilizarea referinţei `this` în următoarele circumstanţe:

1. Pentru rezolvarea ambiguităţilor care apar când numele parametrilor formali ai unei metode coincid cu numele altor membri.
2. Returnarea de către o metodă a unei referinţe la obiectul curent.

```
using System;

class Copil
{
    private string nume;
    private int  varsta;
```

```
    // Parametrii formali au aceleaşi nume cu ale câmpurilor
    public Copil(string nume, int varsta)
    {
        this.nume = nume;          // cazul 1
        this.varsta = varsta;      // cazul 1
    }
    public Copil ModificVarsta(int varsta)
    {
        this.varsta = varsta;      // cazul 1
        return this;               // cazul 2
    }
    public void Afiseaza()
    {
        Console.WriteLine(nume + " " + varsta);
    }
}

class Test_this
{
    static void Main(string[] args)
    {
        Copil c = new Copil("Alin", 17);
        c.Afiseaza();            // Ieşire: Alin 17
        c.ModificVarsta(18);
        c.Afiseaza();            // Ieşire: Alin 18
    }
}
```

Ce s-ar fi întâmplat dacă constructorul clasei s-ar fi definit astfel :

```
public Copil(string nume, int varsta)
{
    nume = nume;
    varsta = varsta;
}
```

Nu este eroare la compilare. Dar parametrul formal **nume** "ascunde" câmpul clasei, astfel încât metoda nu accesează câmpul privat al clasei, ci realizează o inutilă autoatribuire a parametrului formal. Oricare membru al clasei poate fi accesat în metodele clasei astfel: **this.membru**. În acest fel, **this.nume = nume;** realizează o inițializare corectă a câmpului **nume**.

Mai observăm că metoda **ModificVarsta()** returnează o referință la obiectul curent. Cine este de fapt obiectul curent ? Asta nu vom şti, până în momentul în care metoda este apelată: **c.ModificVarsta(18);** Prin urmare, **c** referă obiectul curent şi **this** referă evident acelaşi obiect. De altfel, în momentul apelului, **this** referă deja obiectul alocat cu **new**, pentru că **c** transferă în **this** referința către obiect.

Există şi alte situații în care aveți nevoie de **this**, aşa cum vom vedea mai departe.

Destructorul clasei

Destructorul clasei este o metodă specială care distruge instanțele claselor. Dacă o clasă poate avea mai mulți constructori, în schimb nu poate avea decât un singur destructor. Un destructor se declară astfel:

```
class Avion
{
    // Membrii clasei
    ~Avion()  // Destructor
    {
        // Eliberarea resurselor gestionate de obiect
    }
}
```

În majoritatea covârşitoare a cazurilor, nu veți avea de implementat un destructor, deoareace C# are un ***garbage collector*** care distruge obiectele nefolositoare în mod automat. Veți implementa un destructor doar în situații în care obiectele gestionează resurse care nu sunt controlate de *.NET Framework*.

Nu vom intra în alte detalii care ies din cadrul propus în această lucrare. Este bine să rețineți totuşi câteva informații utile în legătură cu destructorii:

IMPORTANT

- Destructorii au acelaşi nume cu cel al clasei, prefixat cu caracterul ~.
- O clasă poate avea un singur destructor.
- Destructorii nu se apelează niciodată. Ei se invocă în mod automat.
- Constructorii nu au parametri şi nu acceptă modificatori.
- În structuri nu se pot defini destructori.
- Destructori nu pot fi supraîncărcați.

Proprietăți

Proprietățile reprezintă o facilitate importantă a limbajului C#. Sunt membri ai clasei care permit accesul direct la starea obiectului. Starea unui obiect este dată de valorile datelor membre. Cu ajutorul proprietăților veți accesa câmpurile private, ca şi când ar fi fost declarate public, fără ca prin aceasta să se încalce principiul protecției datelor, care cere ca datele membre să fie private. În realitate, proprietățile implementează nişte metode speciale, numite ***accesori***.

Exemplu:

```
using System;

class Copil
{
    private string nume;              // Câmpuri private
    private int varsta;
```

```
    public Copil(string n, int v)      // Constructor
    {
        nume = n;
        varsta = v;
    }

    public string Nume                 // Proprietatea Nume
    {
        get
        {
            return nume;
        }
        set
        {
            nume = value;
        }
    }

    public int Varsta                  // Proprietatea Varsta
    {
        get
        {
            return varsta;
        }
        set
        {
            varsta = value;
        }
    }
}

class TestProprietati
{
    static void Main(string[] args)
    {
        Copil c = new Copil("Valentin", 18);
        Console.WriteLine(c.Nume + " " + c.Varsta); // get
        c.Nume = "Andrei";     // set
        c.Varsta = 20;         // set
        Console.WriteLine(c.Nume + " " + c.Varsta); // get
    }
```

Fiecare proprietate returnează şi eventual setează şi valoarea unui câmp privat. S-au implementat două proprietăţi: `Nume` şi `Varsta`. Toate proprietăţile au tip.

Prototipul proprietății este format din tipul returnat de către proprietate (aici `string`, respectiv `int`) şi numele proprietății (`Nume` şi `Varsta`). Tipul proprietății, este tipul câmpului pe care îl gestionează proprietatea.
Corpul proprietății conține doi accesori: `get` şi `set`. Unul din ei poate să lipsească.

- `get` are următoarele caracteristici:
  - ✓ Nu are parametri
  - ✓ Are tip de retur de acelaşi tip cu al proprietății.
- `set` are următoarele caracteristici:
  - ✓ Are un singur parametru implicit, numit `value`, de acelaşi tip cu proprietatea. `value` reține valoarea câmpului gestionat de proprietate.
  - ✓ Are tipul de retur `void`.

Tipul proprietății — Numele proprietății

```
public string Nume
{
    get                      // get nu are parametri. Returnează
    {                        // o valoare de tipul proprietății
        return nume;
    }
    set                      // set nu are parametri expliciți.
    {                        // Parametrul implicit este value
        nume = value;
    }
}
```

Accesorii pot fi declarați în orice ordine. Nici o altă metodă în afară de `get` şi `set` nu poate exista în corpul proprietății. Accesorii se apelează în mod implicit.

Citirea şi scrierea într-o proprietate se face la fel ca pentru un câmp public:

- Expresia `c.Nume = "Andrei";` atribuie câmpului privat `nume` valoarea `"Andrei"` în felul următor: set se apelează în mod implicit, iar parametrul său implicit `value`, primeşte valoarea `"Andrei"`. În corpul metodei, prin `nume = value;` are loc atribuirea necesară.
- Expresia `c.Nume` (scrisă în contextul unei afişări) returnează valoarea proprietății, deci a câmpului `nume`, astfel: `get` se apelează în mod implicit. Metoda `get` returnază valoarea proprietății, deci a câmpului privat `nume`.

Proprietățile nu încalcă principiul protecției datelor

Aparent, un câmp declarat `public` (o practică descurajată în OOP) se accesează în mod identic cu o proprietate asociată unui câmp privat, iar proprietatea pare că nu protejează câmpul de acțiuni nedorite. Nu este de loc aşa.

În primul rând aveți libertatea să creați proprietăți:

- *Read-Only*, adică proprietăți care nu implementează metoda `get`.
- *Write-Only*, adică proprietăți care nu implementează metoda `set`.
- Nu puteți defini o proprietate din care să lipsească atât `get` cât şi `set`.

În al doilea rând, metoda **set** se poate implementa de asemenea manieră, încât câmpul asociat proprietății (*backing field*) să fie complet protejat de acțiuni nedorite.

Exemplu:

În set puteți plasa orice cod defensiv de protecție a câmpului asociat proprietății:

```
using System;

class Intreg
{
    private int n;              // Câmp privat
    public int N
    {
        get { return n; }

        set
        {   // Cod care protejează n de acțiuni nedorite
            if (value > 1000)  n = 1000;
            else
                if (value < 0) n = 0;
                else n = value;
        }
    }
}

class Test
{
    static void Main(string[] args)
    {
        Intreg i = new Intreg();
        i.N = 1200;     // set
        Console.Write(i.N + " "); // get
        i.N = -10;      // set
        Console.Write(i.N);       // get
    }
}
```

Ieşire: 1000 0

Proprietăți statice

Proprietățile se pot declara **static**. Procedați astfel când doriți să le asociați unor câmpuri statice. Proprietățile statice nu pot accesa alți membrii ai clasei, decât pe cei statici și există independent de existența instanțelor clasei. Se accesează prin numele clasei, la fel ca alți membri statici.

Exemplu:

```
class Numar
{
    private static int nr;         // Câmp static
    public static int Nr           // Proprietate statică
    {
        get { return nr; }
        set { nr = value; }
    }
}

class TestConst
{
    static void Main(string[] args)
    {
        Numar.Nr = 100;    // set
        System.Console.Write(Numar.Nr); // get
    }
}
```

Convenție de notare

Pentru claritate, se obişnuieşte ca numele proprietății să fie acelaşi cu cel al câmpului privat asociat. Diferența este că în timp ce câmpul începe cu literă mică, proprietatea începe cu literă mare. Sfatul nostru este să urmați această convenție.

Proprietăți implementate în mod automat

Aceste tipuri de proprietăți, au următoarele caracteristici:

- Nu se asociază nici unui câmp al clasei.
- Compilatorul alocă memorie pentru valorile furnizate, în funcție de tipul proprietății.
- Accesorii **get** şi **set** nu au corp.
- Nu pot fi *read-only* (nu poate lipsi **get**) şi nici *write-only* (nu poate lipsi **set**).

Exemplu:

```
using System;

class X
{
    public int Valoare      // Proprietate
    {                       // implementată
        get; set;           // în mod automat
    }
}
```

```
class TestAutoPropertie
{
    static void Main(string[] args)
    {
        X x = new X();
        x.Valoare = 10;             // set
        Console.Write(x.Valoare);   // get
    }
}
```

Veți utiliza o proprietate automatică atunci când nu aveți nevoie să impuneți restricții speciale valorilor sale.

Restricțiile proprietăților

- Proprietățile nu definesc o locație de memorie. Din acest motiv, nu pot fi transmise cu modificatorii **out** sau **ref**.
- Nu pot fi supraîncărcate.

Indexatori

Indexatorii permit instanțelor unei clase sau ale unei structuri să fie indexate, la fel ca elementele unui tablou. Indexatorii și proprietățile sunt similari din mai multe puncte de vedere:

- Sintaxă asemănătoare.
- Pot avea unul sau doi accesori.
- Nu definesc o locație de memorie.
- Codul accesorilor poate să fie asociat sau să nu fie cu un câmp al clasei.

Există și **diferențe**:

- O proprietate accesează de regulă un singur câmp privat, în timp ce un indexator poate accesa mai mulți membrii ai clasei.
- Numele unui indexator este întotdeauna **this**.
- Un indexator este întotdeauna un membru de instanță. Din această cauză nu poate fi declarat **static**.

Sintaxă:

```
TipReturnat this[int index]
{
    get
    {
        // Se returnează valoarea precizată prin index
    }
    set
    {
        // Se modifică valoarea precizată prin index
    }
}
```

TipReturnat este tipul de bază al indexării. Parametrul *index* primește valoarea indexului elementului care va fi accesat.

Exemplu:

```
using System;

class Tablou
{
    private int[] a;          // Tablou conținut - Câmp privat
    private int n;            // Dimensiunea tabloului
    private const int MAX = 1000;

    public Tablou(int dim)        // Constructorul clasei
    {
        n = dim;
        a = new int[n];
    }

    public int this[int index]   // Indexator
    {
        get    // Returnează a[index] doar dacă index este
        {      // în intervalul [0, MAX]
            if (index < 0)
                return a[0];
            else
                if (index > MAX)
                    return a[MAX];
                else
                    return a[index];
        }

        set    // Setează a[index] doar dacă index este
        {      // în intervalul [0, MAX]
            if (index < 0)
                a[0] = value;
            else
                if (index > MAX)
                    a[MAX] = value;
                else
                    a[index] = value;
        }
    }
}

class TestIndexer
{
    static void Main(string[] args)
    {
        Tablou t = new Tablou(100);
        for (int i = 0; i < 10; i++)
            t[i] = i;        // set
        t[-1] = 10;          // set
```

```
        for (int i = 0; i < 10; i++)
            Console.Write(t[i] + " ");   // get
    }
}
```

Ieşire:

```
10 1 2 3 4 5 6 7 8 9
```

În exemplul de mai sus se aplică operatorul `[]` referinței `t`, accesându-se în felul acesta elementele tabloului `a`, conținut în clasă.

Dacă o clasă conține ca şi cîmp privat o colecție, iar acea colecție suportă operația de indexare, atunci puteți defini un indexator care vă va permite să accesați elementele colecției cu ajutorul operatorui `[]` aplicat referinței la obiectul de tipul clasei (mai sus, `t[i]`).

Operatori de conversie

În capitolul **3**, paragraful *Conversii între tipuri*, am văzut că între tipurile predefinite se pot face conversii *implicite* sau *explicite*. Conversiile explicite se realizează cu ajutorului operatorului `( )`.

Exemplu:

```
double x = 2.3;
int y = 6;
x = y;          // Conversie implicită
y = (double)x;  // Converise explicită
```

C# vă permite să definiți un tip special de operatori în clase şi structuri, care realizează conversii implicite sau explicite între tipul clasei sau a structurii şi alte tipuri predefinite sau tipuri `class` sau `struct`.

Operatorii de conversie sunt metode statice, introduse cu cuvântul cheie `operator`.

Sintaxă:

```
public static explicit operator Tip1 (Tip2 t)
{
   // Cod care converteşte în mod explicit t spre TipulClasei
}
```

sau

```
public static implicit operator Tip1(Tip2 t)
{
   // Cod care converteşte în mod implicit t spre TipulClasei
}
```

Tip1 este tipul spre care se face conversia. *Tip2* este tipul care se converteşte spre *Tip1*. Unul dintre cele două tipuri trebuie să fie de tipul clasei sau structurii care conţine operatorul de conversie.

Exemplul 1:

Definim doi operatori care realizează conversii explicite de la tipul **Bancnota** spre **int** şi de la **int** spre tipul **Bancnota**:

```
// conversii_explicite.cs
using System;

class Bancnota
{
    private int valoare;

    public Bancnota(int v)   // Constructor
    {
        valoare = v;
    }

    // Operator de conversie explicită de la int la Bancnota
    public static explicit operator Bancnota(int v)
    {
        return new Bancnota(v);
    }

    // Operator de conversie explicită de la Bancnota la int
    public static explicit operator int(Bancnota b)
    {
        return b.valoare;
    }
}

class TestConversieExplicita
{
    static void Main()
    {
        int val = 100;
        Bancnota b = (Bancnota)val;  // Conversie explicită
        Console.WriteLine((int)b);   // Conversie explcită
    }
}
```

Exemplul 2:

Modificăm programul anterior, astfel încât să obţinem operatori de conversii implicite:

```
// conversii_implicite.cs
using System;

class Bancnota
{
    private int valoare;

    public Bancnota(int v)  // Constructor
    {
        valoare = v;
    }

    // Operator de conversie implicită de la int la Bancnota
    public static implicit operator Bancnota(int v)
    {
        return new Bancnota(v);
    }

    // Operator de conversie implicită de la Bancnota la int
    public static implicit operator int(Bancnota b)
    {
        return b.valoare;
    }
}

class TestConversieImplicita
{
    static void Main()
    {
        int val = 100;
        Bancnota b = val;         // Conversie implicită
        Console.WriteLine(b);     // Conversie implicită
    }
}
```

Observaţi că nu a mai fost necesară utilizarea operatorului ( ) - *type cast operator*.

Clase interioare

O clasă interioară (*inner class* sau *nested class*) este o clasă a cărei definiţie se găseşte în interiorul definiţiei altei clase.

1. Pentru a instanţia un obiect de tipul clasei interioare, nu este nevoie de un obiect de tipul clasei exterioare:

Exemplu:

```
using System;

class A          // Clasa container
{
    public class B  // Clasa interioară
    {
        public void F() { Console.Write("F()"); }
    }
}

class TestInnerClass
{
    static void Main()
    {
        A.B obj = new A.B(); // Creează un obiect de tip B
        obj.F();    //Afişează: F();
    }
}
```

2. Metodele clasei interioare pot accesa toți membrii clasei care o conține, prin intermediul operatorului '.' :

Exemplu:

```
using System;

public class Exterior
{
    private int x;
    public Exterior(int y)     // Constructor
    {
        x = y;
    }

    public class Interior
    {
        public void Scrie(Exterior o)
        {
            // Acces permis la membrii clasei Exterior
            System.Console.WriteLine(o.x);
        }
    }
}

public class TestInnerClass
{
    static void Main()
```

```
    {
        Exterior o1 = new Exterior(100);

        // Instanţierea unui obiect de tip Interior
        Exterior.Interior o2 = new Exterior.Interior();
        o2.Scrie(o1);    // Afişează: 100
    }
}
```

Conţinere

Când o clasă are ca membri unul sau mai multe obiecte de tip clasă, vorbim despre ***conţinere*** (*containment*) sau ***compoziţie***.

În figură, ilustrăm clasa **Calculator**. Un calculator are mai multe componente. Acestea sunt instanţe ale altor clase : **Monitor**, **Tastatura**, **Mouse** şi **UnitateCentrala**.

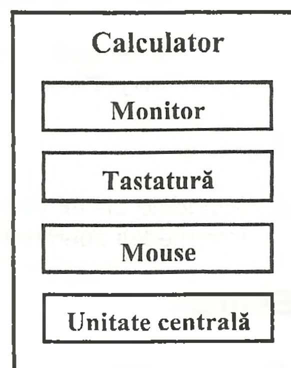


Figura 4.1 *Conţinere*

IMPORTANT
Tipul de relaţie care se modelează în cazul conţinerii este: *"**HAS A**"* (*ARE UN, ARE O*).

Exemplu:

Programul de mai jos defineşte trei clase: **Motor**, **Caroserie**, **Masina**. Clasa **Masina** are ca membri privaţi câte o referinţă spre obiecte de tip **Motor** şi **Caroserie**.

```
class Motor { /* ... */ }

class Caroserie { /* ... */ }

class Masina
{
    private string marca;
    // Maşina ARE O caroserie c şi ARE UN motor m
    private Caroserie c;
    private Motor m;
    // Constructorul clasei
    public Masina(Caroserie c, Motor m, string marca)
    {
        this.c = c;
        this.m = m;
        this.marca = marca;
    }
}
```

```
class TestContinere
{
    public static void Main()
    {
        Caroserie C = new Caroserie();
        Motor M = new Motor();
        Masina mea = new Masina(C, M, "Volvo");
    }
}
```

Clase parțiale

Definițiile claselor, structurilor şi interfețelor se pot împărți în mai multe fişiere sursă. Ca alternativă, puteți fragmenta o clasă în interiorul aceluiaşi fişier. Pentru a separa în blocuri o definiție de clasă, utilizați cuvântul cheie **parțial**.

Exemplu:

```
using System;

partial class A
{
    public void F()
    {
        System.Console.Write ("F() ");
    }
}
partial class A
{
    public void G()
    {
        System.Console.WriteLine("G()");
    }
}

public class TestPartialClass
{
    static void Main()
    {
        A a = new A();
        a.F();
        a.G();
    }
}
```

Ieşirea programului : `F() G()`

Practic, este vorba despre o singură clasă cu numele **A**, a cărei definiţie a fost fragmentată. Compilatorul unifică definiţiile parţiale şi instanţiază în mod corect obiectul `a` de tip `A`.

Mediul *Visual C# 2008* generează în mod curent asemenea definiţii parţiale. Dacă creaţi un proiect de tip *Windows Forms*, veţi constata că definiţia de clasă a formei principale, să-i spunem `Form1`, este fragmentată în două fişiere diferite: *Form1.cs* şi *Form1.Designer.cs*.

Clase sigilate

sealed este un cuvânt cheie. Când modificatorul **sealed** este aplicat unei clase, are efectul de a împiedica alte clase să o moştenească.

Exemplu:

```
sealed class A { /*...*/}

class B : A { /*...*/ }      // Eroare
```

Supraîncărcarea operatorilor

Toţi operatorii unari şi binari au implementări predefinite care sunt disponibile în orice expresie în care sunt implicate tipuri predefinite. C# vă permite să redefiniţi semnificaţia unui operator standard, astfel încât operatorul respectiv să poată fi aplicat instanţelor clasei voastre. În acest fel sporiţi funcţionalitatea claselor.

Ce înţelegem prin supraîncărcarea operatorilor (*operator overloading*)? Este un mecanism prin care instanţele clasei pot fi integrate în expresii aritmetice sau logice în calitate de operanzi, cu utilizarea operatorilor specifici tipurilor predefinite: +, -, *, /, <, >, etc. Când definiţi o metodă de tip operator, de exemplu pentru operaţia de adunare, atunci spunem că "*am supraîncărcat operatorul* +".

Să presupunem că avem o clasă care descrie o fracţie şi dorim să efectuam operaţii cu instanţe de tip `Fractie`, în felul acesta:

```
Fractie f1 = new Fractie();
Fractie f2 = new Fractie();
Fractie f = f1 + f2;
```

În C# operatorii supraîncărcaţi sunt metode statice ai căror parametri sunt operanzii, iar valoarea returnată este rezultatul operaţiei.

Sintaxa de declarare a unui operator pretinde folosirea cuvântului cheie `operator`, urmat de simbolul operatorului care se redefineşte.

Sintaxa

În cazul operatorilor binari:

```
public static Tip operator op(Tip1 operand1, Tip2 operand2)
{
      // operaţii
}
```

În cazul operatorilor unari:

```
public static Tip operator op(Tip operand)
{
      // operaţii
}
```

op este operatorul care se supraâncarcă: +, -, /, -, etc.

Pentru operatorii unari, operandul trebuie să fie de acelaşi tip cu clasa pentru care supraîncărcaţi operatorul. Pentru operatorii binari, cel puţin unul dintre cei doi operanzi trebuie să fie de tipul clasei. `Tip` este tipul de retur al operaţiei şi de regulă este un obiect de tipul clasei.

Supraîncărcarea operatorilor binari

În exemplul de mai jos, clasa **Fracţie**, supraîncarcă operatorii +, ==, !=:

```
// fractie.cs
using System;

public class Fractie
{
    private int numarator;
    private int numitor;
    private static int Cmmdc(int a, int b)
    {
        if (b == 0) return a;
        return Cmmdc(b, a % b);
    }
    public Fractie(int numarator, int numitor) // Constructor
    {
        int div = Cmmdc(numarator, numitor);
        numarator /= div; numitor /= div;
        this.numarator = numarator;
        this.numitor = numitor;
    }

    public static Fractie operator +(Fractie f1, Fractie f2)
    {
        int A = f1.numarator * f2.numitor +
                f2.numarator * f1.numitor;
```

```
        int B = f1.numitor * f2.numitor;
        int cmmdc = Cmmdc(A, B);
        A /= cmmdc;
        B /= cmmdc;
        return new Fractie(A, B);
    }

    public static bool operator ==(Fractie f1, Fractie f2)
    {
        if (f1.numitor == f2.numitor &&
            f1.numarator == f2.numarator)
            return true;
        return false;
    }

    public static bool operator !=(Fractie f1, Fractie f2)
    {
        return !(f1 == f2);
    }

    // Suprascrie Object.ToString()
    public override string ToString()
    {
        String s = numarator.ToString() + "/" +
                   numitor.ToString();
        return s;
    }
}

public class TestFractie
{
    static void Main()
    {
        Fractie a = new Fractie(3, 4);
        Console.WriteLine("a = {0}", a.ToString());
        Fractie b = new Fractie(2, 4);
        Console.WriteLine("b = {0}", b.ToString());

        Fractie c = a + b;  // operator+
        Console.WriteLine("c = a + b = {0}",
                          c.ToString());
        Fractie d = new Fractie(2, 4);

        if (d == b)             // operator==
           Console.WriteLine("d = b = {0}", d);

        if (a != b)             // operator!=
            Console.WriteLine(a + " != " + b);

        a += b;                 // operator +=
```

```
        Console.WriteLine("a = " + a);
    }
}
```

Ieşire:

```
a = 3/4
b = 1/2
c = a + b = 5/4
d = b = 1/2
3/4 != 1/2
a = 5/4
```

De reţinut:

- Operatorul de atribuire = nu se supraîncarcă. El este furnizat în mod implicit de către compilator (c = **a + b**).
- Operatorii unari au un parametru, iar cei binari doi parametri: în ambele cazuri, cel puţin unul dintre operanzi trebuie să fie de tipul clasei sau al structurii.
- Operatorii unari au un singur parametru.
- C# pretinde ca în situaţia în care supraîncărcaţi == atunci trebuie să supraîncărcaţi şi `!=`. Similar, operatorii < şi > trebuie supraîncărcaţi în perechi, la fel ca şi <= cu >=.
- Dacă supraîncărcaţi operatorul +, atunci C# sintetizează automat +=. Aceeaşi regulă este valabilă şi pentru + cu +=, * cu *=, etc.

Metoda ToString()

În clasa **`Fractie`**, am definit metoda **`ToString()`**. De fapt, am redefinit pentru **`Fracţie`**, această funcţie. **`ToString()`** este o metodă a clasei **`object`**, clasa de bază a tuturor claselor. Rolul ei este de a returna o reprezentare a obiectului clasei într-un obiect de tip **`string`**. De câte ori doriţi o conversie particulară spre **`string`** a obiectelor clasei, redefiniţi această metodă moştenită. Pentru aceasta folosiţi cuvântul cheie **`override`**. Observaţi că metoda **`ToString()`** se apelează în mod implicit atunci când transferaţi obiectul într-un context în care se cere un string: apelul `Console.WriteLine("a = " + a);` pune **a** într-un context în care se aşteaptă un string, deci **a** se converteşte la **`string`** prin apelul implicit al metodei **`ToString()`**.

Supraîncărcarea operatorilor unari

Pentru exemplificarea supraîncărcării operatorilor unari definim o clasă simplă:

```
using System;

public class Intreg
{
```

```
    private int n;
    public Intreg(int i)    // Constructor
    {
        n = i;
    }
    // Operatorul de incrementare
    public static Intreg operator ++(Intreg x)
    {
        return new Intreg(++x.n);
    }
    // Operatorul unar -
    public static Intreg operator -(Intreg x)
    {
        return new Intreg(-x.n);
    }
    public override string ToString()
    {
        String s = string.Format("{0}", n);
        return s;
    }
}

public class TestOpUnari
{
    static void Main()
    {
        Intreg x = new Intreg(10);
        x++;      // Incrementare
        Console.WriteLine(x + " " + -x); // Scrie: 11 -11
    }
}
```

NOTĂ:

Nu este nevoie să creaţi două versiuni diferite ale operatorului ++ ca să suporte incrementare prefixată şi sufixată. O singură versiune este suficientă, iar compilatorul are grijă să implementeze diferenţierea între prefixare şi sufixare.

IMPORTANT

Este recomandabil să nu modificaţi operanzii pasaţi metodelor operator. În loc de aceasta, creaţi noi instanţe de tipul valorii de retur şi returnaţi aceste instanţe. Urmând această practică, veţi evita probleme la depanare.

Operatorii care pot fi supraîncărcaţi:

Operatori unari: +, -, !, ~, ++, --, true, false
Operatori binari: +, -, *, /, %, &, |, ^, >>, <<

Operatori relaţionali: == cu !=, < cu >, <= cu >= (aceştia trebuie supraîncărcaţi în perechi).

Acţiuni nepermise la supraîncărcarea operatorilor:

- Crearea unui nou operator (se pot supraîncărca doar operatori predefiniţi).
- Schimbarea sintaxei unui operator.
- Redefinirea modului de lucru a unui operator cu tipuri predefinite.
- Schimbarea precedenţei sau asociativităţii unui operator.

Structuri

Structurile sunt tipuri de date definite de programator, asemănătoare claselor. Se definesc cu ajutorul cuvântului cheie `struct`.

Structurile sunt asemănătoare claselor prin faptul că pot să conţină câmpuri, metode, constructori, proprietăţi, operatori, tipuri imbricate, indexatori.

Diferă de clase în următoarele privinţe:

- Structurile sunt *tipuri valoare*, iar clasele sunt *tipuri referinţă*.
- Nu suportă moştenirea.
- Nu suportă constructori fără parametri.
- Nu au destructori.

Sintaxa:

```
[ModificatorAcces] struct  NumeStructură
{
    // membrii structurii
}
```

Exemplu:

```
using System;

struct Punct
{
    private double x;
    private double y;

    public Punct(double _x, double _y) // Constructor
    {
        x = _x;
        y = _y;
    }

    public double X                   // Proprietatea X
    {
        get { return x; }
```

```
        set { x = value; }
    }
    public double Y                          // Proprietatea Y
    {
        get { return y; }

        set { y = value; }
    }
}

class TestStruct
{
    static void Main(string[] args)
    {
        Punct p1 = new Punct(2.3, 3.5);
        Console.Write("x = {0}, y = {1}", p1.X, p1.Y);
    }
}
```

Structurile sunt recomandabile pentru obiecte mici, aşa cum este un obiect de tip **Punct**, care trebuie instanţiate în număr mare, eventual într-o buclă. Pentru că sunt tipuri valoare, se construiesc pe stiva programului, care se accesează mai rapid decât memoria *Heap*.

Interfeţe

O interfaţă este un tip referinţă care descrie un set de metode, dar nu le implementează. Clasele şi structurile pot implementa interfeţele. Când o clasă implementează o interfaţă, trebuie să implementeze toate metodele acelei interfeţe. În felul acesta, clasa "semnează un contract", pe care se obligă să-l respecte.

Sintaxa minimală:

```
[modificatori] interface NumeInterfaţă
{
    // Corpul interfeţei
}
```

Modificatorii pot fi **public**, **private**, **protected**, **internal**, **protected internal**. Cuvântul cheie **interface** precedă numele interfeţei. Este o practică comună ca numele interfeţei să înceapă cu litera **I**. Exemple: **Icomparable**, **ICloneable**, **IControl**, etc.

Implementarea interfeţelor de către clase

O clasă sau o structură poate implementa una sau mai multe interfeţe, ca mai jos:

```
interface I1 { /*...*/ }
interface I2 { /*...*/ }

class C : I1, I2
{ /*...*/ }
```

Exemplu:

```
using System;

interface IPrintable
{
    void Print(string s);
}

public class Mail : IPrintable
{
    private string s;
    public Mail(string s)       // Constructor
    {
        this.s = s;
    }
    // Implementarea metodei interfeţei (obligatoriu!)
    public void Print(string a)
    {
        Console.WriteLine(s + a);
    }
}

public class TestOpUnari
{
    static void Main()
    {
        Mail m = new Mail("Prietenilor mei ");
        m.Print("Salut!");
    }
}
```

Ieşire:
`Prietenilor mei Salut!`

Ceea ce trebuie să reţineţi despre interfeţe este:

- O interfaţă nu se poate instanţia.
- Interfeţele nu conţin câmpuri, ci doar metode, proprietăţi, evenimente.
- Interfeţele nu conţin implementări ale metodelor.
- Clasele şi structurile pot moşteni (impelmenta) una sau mai multe interfeţe.
- O interfaţă poate la rândul ei să moştenească o altă interfaţă.

Moştenire

Moştenirea este unul dintre cele mai importante şi mai puternice concepte în *Programarea Orientată pe Obiecte*. Moştenirea vă permite să definiţi o nouă clasă care incorporează şi extinde o clasă existentă.

Diagrama de mai jos prezintă o ierarhie de clase bazată pe moştenire. Un animal ***este o*** *fiinţă*, un om ***este o*** fiinţă, un câine ***este un*** animal, un bărbat ***este*** om, ş.a.m.d. Vom generaliza cu afirmaţia următoare:

Moştenirea modelează relaţia ***IS A*** (***ESTE UN, ESTE O***).

Spunem că **`Om`** şi **`Animal`** moştenesc clasa **`Fiinţa`**. Ele sunt ***clase derivate*** din clasa **`Fiinţa`**, iar aceasta din urmă este ***clasă de bază*** pentru **`Om`** şi **`Animal`**. Un obiect de tipul clasei derivate, conţine ca ***subobiect***, un obiect de tipul

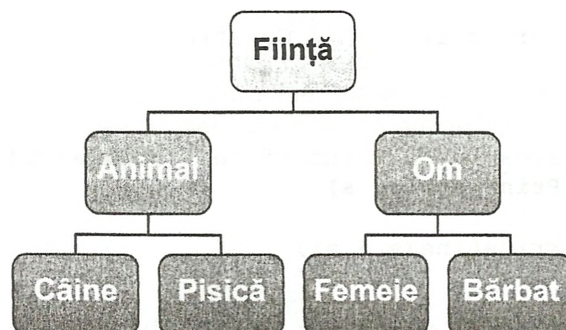


Figura 4.2 *Ierarhie de clase bazată pe moştenire*

clasei de bază. Aşadar, un obiect de tip **`Om`** conţine ca subobiect un obiect de tip **`Fiinţa`**, iar un obiect de tip **`Femeie`**, conţine un subobiect de tip **`Om`**, care la rândul său include un subobiect de tip **`Fiinţa`**.

Specializare şi generalizare

Una dintre cele mai importante relaţii între obiecte în lumea reală este **specializarea**, care poate fi descrisă ca o relaţie ***IS A***. Când spunem că un câine ***este un*** animal, ne gândim de fapt că un câine este un tip specializat de animal. Câinele ***este un*** animal pentru că are toate caracteristicile unui animal, dar specializează aceste caracteristici conform speciei sale. O pisică ***este un*** animal, deci are în comun cu câinele caracteriscticile animalelor (ochi, gură, etc) dar diferă faţă de un câine prin caracteristici specifice pisicilor.

Pe de altă parte, tipul **`Animal`**, **generalizează** tipurile **`Pisică`** şi **`Câine`**.
Aceste relaţii sunt ierarhice. Ele formează un arbore. Urcând în arbore, generalizăm; coborând, specializăm.

Moştenirea presupune atât **specializare** cât şi **generalizare**. În C#, dubla relaţie de specializare şi de generalizare este implementată folosind principiul moştenirii.

Implementarea moştenirii

Fie clasa `A` care moşteneşte clasa `B`. Operatorul `:` exprimă relaţia de moştenire:

```
class B         // B - Clasa de bază
{
    // Membrii clasei B
}

class A : B     // A - Clasa derivată
{
    // Membrii clasei A
}
```

Sintaxa `A : B` se numeşte *specificaţie de clasă de bază* (*base class specification*).

Membrii clasei derivate sunt:
- Membrii definiţi în propria clasă.
- Membrii clasei de bază.

Se spune despre o clasă că ***extinde*** clasa sa de bază, pentru că include membrii clasei de bază, plus orice caracteristică şi funcţionalitate suplimentară furnizată de propria declarare.

Clasa `B` din figura 4.2 are un câmp şi o metodă. Clasa `A`, în dreapta, îşi defineşte proprii membri: un câmp şi o metodă şi în plus moşteneşte clasa `B`, deci are un câmp şi o metodă suplimentare pentru că un obiect de tip `A` include un subobiect de tip `B`.

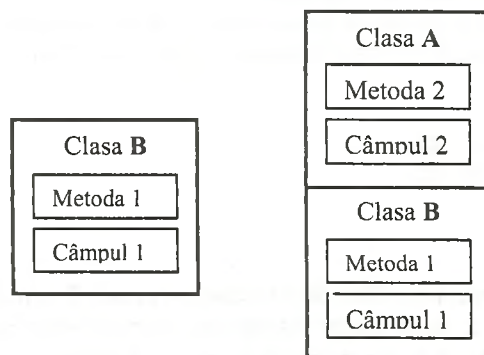


Figura 4.3. ***Clasa* A *moşteneşte clasa* B**

IMPORTANT

În C# toate clasele sunt clase derivate. Oricare clasă derivă în mod direct sau indirect din clasa `object`*. Dacă o clasă nu specifică în mod explicit derivarea dintr-o altă clasă, atunci ea moşteneşte direct clasa* `object`*. În felul acesta,* ***object*** *este baza tuturor ierarhiilor de clase.*

Exemplu:

Cele două declarări ale clasei `Avion` specifică acelaşi lucru: faptul că `Avion` derivă din `object`:

```
// Derivare implicită
// din object
class Avion
{
    // membrii clasei
}
```

```
// Derivare explicită
// din object
class Avion : object
{
    // membrii clasei
}
```

Alte câteva aspecte trebuie reţinute în legătură cu moştenirea în C#:

- **O clasă poate moşteni o singură altă clasă şi oricâte interfeţe**
  În C# o clasă poate moşteni o singură clasă de bază. Acest tip de moştenire se numeşte ***moştenire singulară*** (*single inheritance*).
  O clasă C# poate moşteni în schimb, oricâte interfeţe.
  Fie clasele `A` şi `B` şi interfeţele `I1`, `I2`, `I3`. Dacă `A` moşteneşte `B` şi interfeţele `I1`, `I2`, `I3`, atunci în lista care specifică relaţia de moştenire, clasa `B` trebuie să fie precizată înaintea interfeţelor:

```
// Corect
class A : B, I1, I2, I3
{
}
```

```
// Incorect
class A : I1, B, I2, I3
{
}
```

- **C# nu admite *moştenirea multiplă***
  Moştenirea multiplă (*multiple inheritance*) are loc atunci când o clasă are mai multe clase de bază. Dintre limbajele importante, doar C++ suportă acest tip de moştenire.

```
// Incorect în C#
class A : B, C, D
{
}
```

- **Înălţimea arborelui relaţiilor de moştenire poate fi oricât de mare**
  Mai jos, `A`, `B`, şi `C` sunt clase. Rădăcina ierarhiei este `object`. Din `object` derivă `D`, din `D` derivă `C`, din `C` derivă `B`, iar din `B` derivă `A`:

```
// Corect
class C : D { }
class B : C { }
class A : B { }
```

Accesarea membrilor moşteniţi

O instanţă a unei clase moşteneşte toţi membrii clasei de bază, cu excepţia constructorilor. Totuşi accesul la membrii clasei de bază poate fi restricţionat, cu ajutorul *modificatorilor de acces*. Aceştia sunt: **public**, **private**, **protected**, **internal** şi **protected internal** Programul următor, testează modul în care modificatorii de acces controlează modul de acces la membrii clasei de bază.

```
using System;

class Animal     // Clasa de bază
{
    private string hrana;          // Câmpuri
    protected int speranta_viata;
    public string Hrana            // Proprietate
    {
        set { hrana = value; }
        get { return hrana; }
    }
    protected void SeHraneste()    // Metodă
    {
        Console.WriteLine("Animalul se hraneste");
    }
}

class Pisica : Animal        // Pisica - clasă derivată
{
    private string rasa;
    public string Rasa
    {
        set { rasa = value; }
        get { return rasa;  }
    }
    public void Toarce()
    {
        /* Membrii protejaţi şi cei publici ai clasei de bază
           pot fi accesaţi din metodele clasei derivate */

     // hrana = "lapte";        // Eroare! -câmp privat
        speranta_viata = 12;    // Corect! -câmp protejat
        Hrana = "lapte";        // Corect! -proprietate publică
        SeHraneste();           // Corect! -metodă protejată
        Console.WriteLine("Pisica toarce");
    }
}
```

```
public class TestAccesMembri
{
    static void Main()
    {
        Animal a = new Animal();
    //  a.hrana = "carne";         // Eroare! -câmp privat
    //  a.speranta_viata = 10;     // Eroare! -câmp protejat
        a.Hrana = "carne";       // Corect! -proprietate publică
    //  a.SeHraneste();          // Eroare! -metodă protejată

        /*  Toţi membrii clasei Animal sunt membri ai clasei
            Pisica. Membrii private şi protected nu pot fi
            accesaţi din exteriorul clasei Pisica */

        Pisica p = new Pisica();
    //  p.hrana = "lapte";         // Eroare! -câmp privat
    //  p.speranta_viata = 12;     // Eroare! -câmp protejat
        p.Hrana = "lapte";       // Corect! -proprietate publică
    //  p.SeHraneste();          // Eroare! -metodă protejată
    //  p.rasa = "siameza";      // Eroare! -câmp privat
        p.Rasa = "Siameza";      // Corect! -proprietate publică
        p.Toarce();              // Corect! -metodă publică
    }
}
```

Ieşire:

```
Animalul se hraneste
Pisica toarce
```

Se desprind următoarele reguli:

- Metodele unei clase au acces la toți membrii declarați în acea clasă, indiferent de nivelul de protecție a membrilor.
- Metodele clasei derivate pot accesa membrii publici şi pe cei protejați ai clasei de bază.
- Metodele clasei derivate nu pot accesa membrii privați ai clasei de bază.
- Din exteriorul clasei derivate şi a celei de bază, se pot accesa numai membrii publici ai clasei de bază. Membrii protejați se comportă în acest caz la fel ca cei privați.

Membrii unei clase marcați cu modificatorii `internal`, sunt vizibili (accesibili) pentru toate clasele din acelaşi fişier sau din acelaşi *assembly*.
Membrii unei clase marcați protected internal sunt vizibili tuturor care o moştenesc şi în plus, tuturor claselor din acelaşi *assembly*.

Constructorii claselor derivate

Constructorii clasei de bază nu se moştenesc. Constructorul clasei derivate apelează în schimb constructorul bazei, pentru a construi porţiunea din obiect specifică bazei.

Exemplu:

```
using System;

class Baza
{
    public Baza()
    {
        Console.WriteLine("Constructor Baza");
    }
}

class Derivat : Baza
{
    public Derivat()
    {
        Console.WriteLine("Constructor Derivat");
    }
}

public class TestConstructor
{
    static void Main()
    {
        Derivat d;
        d = new Derivat(); // Apelul constructorului clasei
    }                      // derivate
}
```

Ieşire:

```
Constructor Baza
Constructor Derivat
```

Exemplul de mai sus pune în evidenţă faptul că la construirea unei instanţe a clasei, constructorul `Derivat()` execută constructorul clasei `Baza` înaintea executării propriului cod. Are loc un *apel implicit* al constructoului fără parametri **Baza()**.

Limbajul C# permite şi **apelul explicit** al constructorului bazei. Sintaxa corespunzătoare este:

```
public Derivat() : base()
{
}
```

Dacă constructorul bazei are parametri, atunci *apelul implicit* nu poate avea loc. Constructorul derivat trebuie să invoce în mod explicit constructorul bazei, furnizându-i şi argumentele corespunzătoare:
Exemplu:

```
using System;

class Baza
{
    private int x;
    public Baza(int y)
    {
        x = y;
    }
}

class Derivat : Baza
{
    private char c;
    public Derivat(char a, int b) : base(b)
    {
        c = a;
    }
}

public class TestConstructor
{
    static void Main()
    {
        Derivat d = new Derivat('T', 10);
    }
}
```

Constructorul clasei derivate trebuie să aibă suficienţi parametri pentru a iniţializa şi câmpurile bazei.

Cum se construieşte o instanţă a clasei derivate ?
Când se invocă constructorul clasei derivate, ordinea construcţiei instanţei este:

1. Iniţializarea membrilor de tip instanţă a clasei derivate.
2. Apelul constructorului clasei de bază.
3. Executarea corpului constructorui clasei derivate.

În cazul în care lanţul ierarhic al moştenirii conţine mai multe clase, atunci fiecare constructor îşi execută mai întâi constructorul bazei sale înaintea propriului corp.

Membrii ascunşi

Dacă în clasa derivată aveţi un câmp cu acelaşi nume cu al unuia din clasa de bază, sau aveţi o metodă cu aceeaşi *signatură* (acelaşi nume şi aceeaşi listă de parametri formali) cu a uneia din clasa de bază, atunci numele membrilor din clasa de bază sunt ascunse metodelor clasei derivate. Pentru a accesa membrii ascunşi ai bazei, se întrebuinţează **new** în faţa membrilor bazei şi cuvântul cheie **base**, pentru accesarea membrilor ascunşi.

Exemplu:

```
using System;

class Baza
{
    public int camp = 10;
    public void Metoda()
    {
       Console.WriteLine("Baza.Metoda()");
    }
}

class Derivat : Baza
{
    new public int camp = 20;
    new public void Metoda()
    {
        base.Metoda();
        Console.WriteLine("Derivat.Metoda() " + base.camp);
    }
}

public class TestMembriAscunsi
{
    static void Main()
    {
        Derivat d = new Derivat();
        d.Metoda();
    }
}
```

Ieşire:

```
Baza.Metoda()
Derivat.Metoda() 10
Baza.Metoda()
```

Polimorfism

Polimorfismul este unul dintre conceptele fundamentale ale programării orientate pe obiecte. Reprezintă caracteristica unei entități de a se comporta în moduri diferite, în funcție de context. În particular, este caracteristica unei variabile referință de a putea referi obiecte de tipuri diferite. C# admite polimorfismul bazat pe moştenire. Acest tip de polimorfism vă permite să invocați *runtime* metode ale claselor derivate cu ajutorul unei referințe la clasa de bază.

Conversia referințelor

Dacă aveți o referință la un obiect al clasei derivate, puteți obține o referință la partea de bază a obiectului, folosind operatorul de conversie () ca mai jos:

```
class Baza
{
    public void Metoda()
    {
       System.Console.WriteLine("Baza.Metoda()");
    }
}

class Derivat : Baza
{
    public void Metoda() // sau new public void Metoda()
    {
        System.Console.WriteLine("Derivat.Metoda()");
    }
}

public class TestConversieRef
{
    static void Main()
    {
        Derivat d = new Derivat();
        d.Metoda();

        // Upcast
        Baza b = (Baza)d;
        b.Metoda();
    }
}
```

Ieşire:

```
Derivat.Metoda()
Baza.Metoda()
```

Se constată că referința la partea de bază a obiectului nu "poate vedea" restul obiectului clasei derivate, deoarece "priveşte" printr-o referinţă `b` la clasa de bază. Prin `b` veţi putea să invocaţi numai `Metoda()` clasei de bază.

Conversia unei referinţe a unui obiect derivat spre o referinţă la clasa de bază se numeşte ***upcast***.

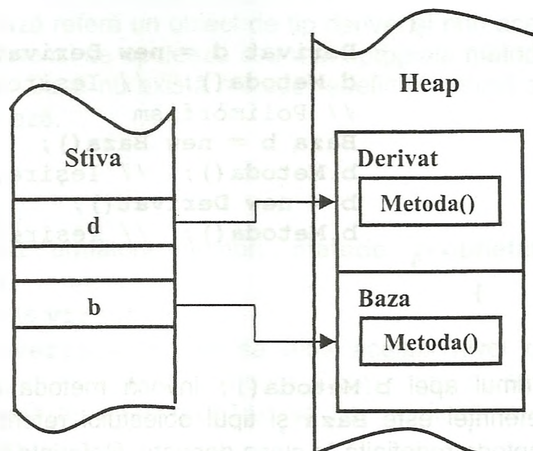


Figura 4.4. *Conversia referinţelor*

Metode virtuale

În paragraful anterior am arătat că într-o clasă derivată puteţi defini o metodă cu acelaşi prototip cu al unei metode din clasa de bază, dar că o referinţă la un obiect de tipul clasei de bază nu poate invoca noua metodă definită cu **new**.

Pentru ca o referinţă la un obiect al bazei să poată accesa membri ai obiectelor derivate, altfel spus, pentru a obţine polimorfism bazat pe moştenire, este nevoie de următoarele:

- Clasa de bază declară acel membru **virtual**.
- Clasa derivată redefineşte acel membru, cu exact aceeaşi semnătură şi tip de retur, adăugând cuvântul cheie **override**, înaintea tipului de retur.

Exemplu:

```
using System;
namespace Polimorfism
{
    class Baza
    {
        virtual public void Metoda()
        {
            Console.WriteLine("Metoda() din Baza");
        }
    }
    class Derivat : Baza
    {
        override public void Metoda()
        {
            Console.WriteLine("Metoda() din Derivat");
        }
    }
    class TestPolimorfism
    {
        static void Main(string[] args)
```

```
        {
            Derivat d = new Derivat();
            d.Metoda();  // Ieşire: "Metoda() din Derivat"
            // Polimorfism
            Baza b = new Baza();
            b.Metoda();  // Ieşire: "Metoda() din Baza"
            b = new Derivat();
            b.Metoda();  // Ieşire: "Metoda() din Derivat"
        }
    }
}
```

Primul apel `b.Metoda();` invocă metoda din clasa de bază, deoarece tipul referinței este **Baza** şi tipul obiectului referit este **Baza**. Al doilea apel invocă metoda redefinită în clasa derivată. Referința **b** este în continuare de tip **Baza**, dar tipul obiectul referit este de tipul clasei derivate. Acesta este manifestarea polimorfismului bazat pe moştenire.

În cazul unui lanț de derivare care porneşte de la o clasă de bază cu metode virtuale, în clasele derivate aveți opțiunea de a redefini sau nu acele metode.

Exemplu:

```
using System;
class A
{
    virtual public void F()
    { Console.WriteLine("F() din A"); }
}

class B : A { }  // B nu redefineşte F()
class C : B
{
    override public void F()
    { Console.WriteLine("F() din C"); }
}

class D : C { }   // D nu redefineşte F()

class Test
{
    static void Main(string[] args)
    {
        A a = new B();  // a referă un obiect de tip B
        a.F();          // Ieşire: "F() din A"
        a = new C();    // a referă un obiect de tip C
        a.F();          // Ieşire: "F() din C"
        a = new D();    // a referă un obiect de tip D
        a.F();          // Ieşire: "F() din C"
    }
}
```

Dacă o referinţă de tipul clasei de bază referă un obiect de tip derivat şi prin acea referinţă invocaţi o metodă virtuală, atunci se apelează cea mai apropiată metodă **override** definită pe lanţul ierarhic. Dacă nu există metode redefinite, atunci se invocă metoda virtuală din clasa de bază.

Restricţii

Reţinem următoarele aspecte:

- Se pot declara **virtual** următorii membri: **metode**, **proprietăţi**, **evenimente** şi **indexatori**.
- Câmpurile nu pot fi declarate **virtual**.
- Metodele redefinite cu **override** trebuie să aibă acelaşi nivel de accesibiliate cu metodele virtuale corespunzătoare. De exemplu nu este corect ca o metodă virtuală declarată **public** să fie redefinită cu o metodă declarată **private**.
- Nu se pot redefini metodele *non-virtuale*.
- Nu pot fi redefini metodele statice.

Modificatorul sealed

Aplicat unei clase, modificatorul **sealed** împiedică alte clase să o moştenească. **sealed** poate fi aplicat şi unei metode sau unei proprietăţi care redefineşte o metodă sau o proprietate a clasei de bază. În această situaţie, se permite altor clase să moştenească clasa care are un membru **sealed**, dar metoda sau proprietatea **sealed** nu mai poate fi redefinită în clasele derivate.

Exemplu:

```
class A
{
    protected virtual void F() { /*...*/ }
    protected virtual void G() {  /*...*/ }
}

class B : A
{
    sealed protected override void F() { /*...*/ }
    protected override void G() { /*...*/ }
}
class D : B   // Corect. B poate fi moştenită
{
    protected override void F() { /*...*/ }  // Eroare !
    protected override void G() { /*...*/ }  // Corect
}
```

Când se aplică unei metode sau unei proprietăţi, modificatorul **sealed** trebuie să fie folosit întotdeauna cu **override**.

Utilitatea polimorfismului

În practică se operează deseori cu colecții de obiecte. Mai precis, cu colecții de referințe la obiecte. Dacă obiectele sunt legate printr-o relație de moştenire având o clasă de bază comună, atunci nu trebuie ca ele să fie de acelaşi tip. Dacă toate obiectele redefinesc o metodă virtuală a clasei de bază, atunci puteți invoca această metodă pentru fiecare obiect.

```
using System;
class LimbaVorbita        // Clasa de bază
{
    virtual public void SeVorbeste() { } // nu va fi invocată
}
class Engleza : LimbaVorbita
{
    override public void SeVorbeste()
    { Console.WriteLine("Engleza"); }
}
class Franceza : LimbaVorbita
{
    override public void SeVorbeste()
    { Console.WriteLine("Franceza"); }
}
class Spaniola : LimbaVorbita
{
    override public void SeVorbeste()
    { Console.WriteLine("Spaniola"); }
}
class Catalana : Spaniola
{
    override public void SeVorbeste()
    { Console.WriteLine("Catalana"); }

}
class Test
{
    static void Main(string[] args)
    {
        LimbaVorbita[] L = new LimbaVorbita[4];
        L[0] = new Engleza();  L[1] = new Franceza();
        L[2] = new Spaniola(); L[3] = new Catalana();

        foreach (LimbaVorbita lv in L)
            lv.SeVorbeste();
    }
}
```

Ieşire:

```
Engleza
Franceza
Spaniola
Catalana
```

Tabloul **LimbaVorbita** reține referințe ale unor obiecte de tipuri diferite, dar care sunt legate prin relație de moştenire, având ca bază a ierarhiei clasa **LimbaVorbita**. Dacă relația de moştenire nu ar exista, atunci referințele ar fi trebuit să fie de acelaşi tip. Programul implementează polimorfismul. Interfața de apelare unică (**lv.SeVorbeste()**) care se utilizează pentru fiecare obiect, duce la acțiuni specifice, în funcție de obiectul referit.

Rezumatul capitolului

- Clasele sunt entități logice care modelează obiecte din lumea reală sau obiecte abstracte. Clasele încapsulează **date** şi **metode** care operează cu aceste date.
- **Constructorii** unei clase sunt metode speciale care au rolul de a creea, a construi obiecte.
- **Destructorul** clasei este o metodă specială care distruge instanțele claselor.
- **Proprietățile** sunt membri ai clasei cu ajutorul cărora se accesează câmpurile private ca şi când ar fi fost declarate public, fără ca prin aceasta să se încalce principiul protecției datelor.
- ***Supraîncărcarea operatorilor*** este mecanismul prin care instanțele clasei pot fi integrate în expresii aritmetice sau logice în calitate de operanzi, cu utilizarea operatorilor specifici tipurilor predefinite: **+**, **-**, *****, **/**, **<**, **>**, etc.
- **Moştenirea** este mecanismul care permite să să se definească o nouă clasă care incorporează şi extinde o clasă existentă.
- **Polimorfismul** este caracteristica unei variabile referință de a putea referi obiecte de tipuri diferite. C# admite polimorfism bazat pe relația de moştenire.

Întrebări şi exerciții

1. Care este diferența dintre un obiect şi o clasă ? Dar dintre o referință şi un obiect ?
2. Indicați principalele deosebiri care există între membrii statici şi cei nestatici ai unei clase.
3. Ce relație se modelează la **moştenire** ? Dar la ***conținere*** ?
4. Implementați o clasă cu numele **Persoana** şi o a doua clasă cu numele **Elev**. Clasele trebuie să aibă constructori, câmpuri private şi proprietăți.
5. Scrieți o clasă cu numele **BigNumber** care implementează operații cu numere mari. Supraîncărcați cel puțin operatorii de adunare, scadere şi înmulțire.

Capitolul 5

Trăsături esenţiale ale limbajului C#

Acest capitol prezintă delegările, evenimentele, genericele, colecţiile, mecanismul de tratare a excepţiilor în C#, un subcapitol dedicat manevrării stringurilor şi un paragraf care se ocupă de operaţiide intrare şi ieşire cu fişiere text.

Delegări

O delegare este un tip referinţă, utilizat să încapsuleze o listă ordonată de metode cu aceeaşi semnătură şi acelaşi tip de retur. Lista de metode se numeşte *lista de invocare*. Când un delegat este invocat, el apelează toate metodele din lista de invocare. O delegare cu o singură metodă în lista sa este similară cu un pointer la funcţii în C++, însă o delegare este un tip referinţă şi oferă siguranţa tipurilor (*type-safe*). Capacitatea unei delegări de a invoca mai multe metode se numeşte ***multicasting***.

Declarare

O delegare este un tip, aşa cum şi clasele sunt tipuri. Un *tip delegare* trebuie declarat înainte de crearea obiectelor de tipul său. Declararea este creată cu cuvântul cheie `delegate`, urmat de tipul de retur şi de *signatura* metodelor pe care delegarea le acceptă.
Exemplu:

```
delegate void DelegareaMea(int x);
```

Expresia declară tipul delegat `DelegareaMea`. Obiectele de tipul acesta vor accepta numai metode cu un singur parametru de tip `int` şi cu tipul de retur `void`. Tipurile delegat nu au corp.

Crearea obiectelor delegare

O delegare este un tip referinţă. Un asemenea tip, conţine referinţa spre obiect, şi obiectul propriuzis.

Referinţele (aici, referinţa se numeşte `d`) se declară simplu:

```
DelegareaMea d;
```

Obiectele de tip delegare se creează în două moduri :

1. Cu sintaxa specifică instanţierii obiectelor:

```
    d = new DelegareaMea(ReferinţăObiect.Metodă);
```
sau
```
    d = new DelegareaMea(NumeClasă.MetodăStatică);
```

Este important de reţinut că delegările pot încapsula atât metode de instanţă cât şi metode statice.

2. Cu sintaxa simplificată, care necesită doar precizarea metodelor ataşate delegării:

```
    d = ReferinţăObiect.Metodă;
```
sau
```
    d = NumeClasă.MetodăStatică;
```

Invocarea metodelor ataşate unei delagări

Prezentăm un exemplu care declară un tip delegat şi o referinţă la cest tip. Apoi ataşează delegării cateva metode şi în cel din urmă le invocă.

```
using System;

class Simplu
{
    // Metodă de instanţă care potriveşte delegării
    public void F(string s)
    {
        Console.WriteLine(s + "F() ");
    }
    // Metodă statică care potriveşte delegării
    public static void G(string s)
    {
        Console.WriteLine(s + "G() ");
    }
}

// Declară tipul delegat Del
delegate void Del(string s);

class Program
{
    static void Main()
    {
        Del d;     // Declară o referinţa de tip Del
        Simplu s = new Simplu();

        // Acum ataşăm metode delegării d
        d = s.F;          // Lista de invocare: F()
        d += Simplu.G;    // Lista de invocare: F(), G()
        d += Simplu.G;    // Lista de invocare: F(), G(), G()
```

```
        // Delegarea d invocă acum toate metodele din listă
        d("Prima invocare: ");

        d -= s.F;        // Lista de invocare: G()
        d("A doua invocare: ");
    }
}
```

Ieşire:

```
Prima invocare: F()
Prima invocare: G()
Prima invocare: G()
A doua invocare: G()
A doua invocare: G()
```

Exemplul anterior descrie modul în care puteţi atribui o metodă listei de invocare a delegării, cu operatorul de atribuire =, apoi puteţi adăuga sau elimina alte metode cu acelaşi prototip cu operatorii += sau -=.

Invocarea delegărilor cu tipuri de retur

Dacă o delegare are o valoare de retur şi mai mult de o metodă în lista de invocare, atunci valoarea care se returnează în urma invocării delegării este valoarea returnată de ultima metodă din lista de invocare. Valorile de retur ale tuturor celorlalte metode din listă sunt ignorate.
Exemplu:

```
using System;

// Declară tipul delegat DelTest
delegate int DelTest();

class Test
{
    private int x = 0;
    // Metode cu acelaşi prototip cu al delegării
    public int F()
    {
        x += 2;
        return x;
    }
    public int G()
    {
        x += 6;
        return x;
    }
}

class Program
{
```

```
static void Main()
{
    Test s = new Test();
    DelTest d;      // Referinţă la tipul delegat

    // Ataşăm metode delegării d
    d = s.F;      // Lista de invocare: F()
    d += s.G;     // Lista de invocare: F(), G()
    d += s.F;     // Lista de invocare: F(), G(), F()
    Console.WriteLine(d());  // Afişeaza: 10
}
}
```

Evenimente

Aplicaţiile cu interfaţă grafică cu utilizatorul sunt sensibile la evenimente ca: click cu mouse-ul pe suprafaţa unei ferestre, apăsarea unei taste, deplasarea mouse-ului deasupra unui control, etc. Sistemul de operare înştiinţează fereastra activă despre apariţia unei acţiuni, iar programatorul poate decide dacă va trata acest eveniment. Evenimentele pot avea o mare diversitate şi nu sunt întotdeauna cauzate de o acţiune directă a utilizatorului aplicaţiei; de exemplu curgerea unui interval de timp, terminarea copierii unor fişiere, primirea unui mail.

> Evenimentele C# permit unei clase sau un obiect să notifice, să înştiinţeze alte clase sau obiecte că ceva s-a întâmplat.

În terminologia specifică, clasa care semnalează evenimentul se numeşte *publisher*, iar clasele care sunt informate despre faptul că a avut loc evenimentul se numesc *subscribers*. Clasele care subscriu evenimentului (clasele *subscriber*) definesc metode pentru tratarea acestui eveniment (*event handler*-e).

Există o mare asemănare între delegări şi evenimente.

Un eveniment este un membru public al clasei care publică evenimentul. Atunci când are loc o acţiune, acest membru al clasei *publisher* se activează, invocând toate metodele care au subscris evenimentului. Activarea se numeşte *declanşare* (*firing the event*).

Lucrul cu evenimente

Pentru utilizarea evenimentelor, programatorul trebuie să scrie cod după cum urmează:

1. **Declară un tip delegat**. Această declarare poate avea loc în clasa *publisher* sau în afara oricărei clase, pentru că este o declarare de tip. Evenimentul şi *handler*-ele de evenimente trebuie să aibă o semnătură şi un tip de retur identice.

2. **Declară evenimentul**. Evenimentul se declară ca membru public în clasa *publisher*, cu ajutorul cuvântului cheie **event**,. El depozitează şi invocă lista de *handler*-e.
3. **Scrie cod care declanşează evenimentul**. Codul se scrie în clasa *publisher*. Apelarea evenimentului duce la invocarea tuturor metodelor *handler* înregistrate cu acest eveniment.
4. **Declară *event handler*-ele**. Se declară în clasele *subscriber* şi sunt metode (*event handler*-e) care se execută când evenimentul se declanşează.
5. **Înregistrează handler**-ele. Aceasta presupune conectarea evenimentului la metodele *handler*. Codul se scrie în clasele subscriber sau în alte clase, dar nu în *publisher*.

Este important de reţinut că prin mecanismul de semnalare şi de tratare a evenimentelor, se realizează o decuplare a obiectelor de tip *publisher* de obiectele de tip *subscriber*. Clasa *publisher* „nu ştie" nimic despre obiectele sau clasele subscriber care vor fi notificate.

Programul care urmează, declară tipul delegat **Timp**. Metodele *handler* trebuie să potrivească acestui tip, adică să nu aibă parametri formali şi tipul de retur să fie **void**. Clasa **Publisher** (evident, poate fi oricare alt nume), declară evenimentul ca şi câmp public cu numele **eveniment**, de tipul **Timp**. Metoda **Declanseaza()** declanşează evenimentul prin apelul **eveniment()**. De fapt, evenimentul ca atare este trecerea a 3 secunde, iar **eveniment()** declanşează apelul metodelor *handler* din lista de invocare a evenimentului. Programul declară două clase subscriber, **A** şi **B**, care definesc câte o metodă de tratare a evenimentului care se semnalează în clasa **Subscriber**. Înregistrarea *handler*-elor are loc în **Main()**, dar s-ar fi putut face şi în metode ale claselor *subscriber*.

```
// event.cs
using System;

public delegate void Timp(); // Declararea tipului delegat

// Publisher nu stie nimic despre obiectele pe care le va
// notifica sau despre metodele inregistrate evenimentului
public class Publisher
{
    public event Timp eveniment; // Declararea evenimentului

    public void Declanseaza()
    {
        while (true)
        {
            // Execuţia programului se întrerupe 3 secunde
            System.Threading.Thread.Sleep(3000);

            // Ne asigurăm că există metode înregistrate
            if (eveniment != null)
                eveniment(); // Declanşează evenimentul
```

```
        }                          // o dată la trei secunde
    }
}

// Clase Subscriber
class A
{
    public void HandlerA()
    {
        Console.WriteLine("Obiect A, notificat la {0}",
                                DateTime.Now);
    }
}

class B
{
    public void HandlerB()
    {
        Console.WriteLine("Obiect B, notificat la {0}",
                                DateTime.Now);
    }
}

class Test
{
    static void Main()
    {
        Publisher p = new Publisher();
        // Obiectele a si b vor fi notificate la declanşarea
        // unui eveniment
        A a = new A();
        B b = new B();

        // Clasele A şi B subscriu acestui eveniment
        p.eveniment += a.HandlerA; // Înregistrarea metodelor
        p.eveniment += b.HandlerB; // handler

        // Apelează metoda care declanşează evenimentul
        p.Declanseaza();
    }
}
```

Programul afişează la fiecare trei secunde, câte două linii:

```
Obiect A, notificat la 12.06.2008 10:13:22
Obiect B, notificat la 12.06.2008 10:13:22
Obiect A, notificat la 12.06.2008 10:13:25
Obiect B, notificat la 12.06.2008 10:13:25
...
```

Pentru determinarea datei şi a orei curente, se utilizeată proprietatea **Now**, a clasei **System.DateTime**.

IMPORTANT
Înregistrarea metodelor ataşate unui eveniment se face cu operatorul +=, iar îndepărtarea lor din lista de invocare, se face cu -=.

Publicarea evenimentelor în mod specific .NET

Tratarea evenimentelor presupune existenţa unui tip delegat. Acesta poate fi creat de dumneavoastră, însă o mai bună alternativă este folosirea tipului delegat predefinit al platformei .NET.

Când evenimentul nu transmite date handler-elor

```
public
delegate void EventHandler(object sender, EventArgs e);
```

sender este o referinţă la obiectul (de tip *publisher*) care declanşează evenimentul, iar **e** este o referinţă la un obiect de tip **EventArgs**. Declararea de mai sus nu trebuie făcută în program, pentru că o face platforma .NET. Ceea ce aveţi de făcut în programul **events.cs**, este să ştergeţi linia care declară tipul delegat **Timp** şi să vă declaraţi un eveniment în clasa *Publisher*, astfel:

```
public event EventHandler eveniment;
```

Handler-ele trebuie să aibă desigur, aceeaşi semnătură şi tip de retur, cu a tipului **EventHandler**:

```
public void HandlerA(object sender, EventArgs e)
```
şi
```
public void HandlerB(object sender, EventArgs e)
```

O ultimă modificare pe care o faceţi, este înlocuirea apelului **eveniment();** cu apelul **eveniment(this, new EventArgs());**

Când evenimentul transmite date handler-elor

Obiectul de tip **EventArgs** nu conţine în realitate date utile şi este folosit atunci când un eveniment nu transmite date de stare unui *event handler*. Dacă doriţi să transmiteţi *handler*-elor informaţii suplimentare despre eveniment, puteţi crea o clasă, de exemplu **MyEventArgs**, care menţine aceste informaţii. Această clasă trebuie să fie derivată (să moştenească) clasa **EventHandler**, aşa cum se vede în programul care urmează. În această situaţie, veţi crea un tip delegat cu o semnătură compatibilă **.NET**. Vom rescrie programul anterior. Tipul delegat se va declara ca mai jos:

```
public delegate void Timp(object sender, MyEventArgs ev);
```

sender este o referinţă la **object**, deci poate referi obiecte de orice tip, inclusiv ***Publisher***. În apelul **eveniment(this, ev);** argumentele sunt: referinţa **this** la obiectul de tip ***Publisher*** şi referinţa **ev** la un obiect de tip **MyEventArgs**.

```
using System;

public class MyEventArgs : EventArgs
{
    private DateTime momentul;    // Câmp
    public DateTime Momentul      // Proprietate
    {
        set
        {
            momentul = value;
        }
        get
        {
            return this.momentul;
        }
    }
}

// Declară tipul delegat cu prototipul cerut de .Net
public delegate void Timp(object sender, MyEventArgs ev);

public class Publisher
{
    public event Timp eveniment;     // Declar evenimentul

    public void Declanseaza()
    {
        while (true)
        {
            // Execuţia programului se întrerupe 3 secunde
            System.Threading.Thread.Sleep(3000);
            if (eveniment != null)  // Ne asigurăm ca există
            {                       // metode înregistrate
                MyEventArgs ev = new MyEventArgs();
                ev.Momentul = DateTime.Now;
                // Declanşează invocarea handler-elor
                   eveniment(this, ev);
            }
        }
    }
}

// Clase Subscriber
```

```
class A
{
    public void HandlerA(object sender, MyEventArgs e)
    {
        Console.WriteLine("Obiect A, notificat la {0}",
                          e.Momentul);
    }
}

class B
{
    public void HandlerB(object sender, MyEventArgs e)
    {
        Console.WriteLine("Obiect B, notificat la {0}",
                          e.Momentul);
    }
}

class Test
{
    static void Main()
    {
        Publisher m = new Publisher();
        A a = new A();  // Obiectele a si b vor fi notificate
        B b = new B();  // la aparitia evenimentului

        // Înregistrarea metodelor handler
        m.eveniment += a.HandlerA;
        m.eveniment += b.HandlerB;

        m.Declanseaza();
    }
}
```

Ieşirea programului este identică cu cea a programului anterior.

De reținut:

Dacă vreți să scrieți cod eficient, veți trata evenimentele folosind clasele de bază ***EventHandler*** şi ***EventArgs***, iar delegările vor avea prototipul compatibil **.NET**, chiar dacă C# admite orice model delegat.

Generice

Programarea Generică sau programarea cu şabloane este un stil de programare diferit de *Programarea Orientată pe Obiecte*. Au în comun abstractizarea datelor şi reutilizarea codului, dar abordările sunt diferite. În timp ce

OOP încapsulează în acelaşi obiect date care reflectă starea obiectului, împreună cu metode care descriu capabilităţile lui, scopul programării generice este scrierea de cod care să fie independent de tipurile de date.

Clase generice

Să presupunem că aţi definit o clasă **Stiva** ca mai jos:

```
public class Stiva
{
    private int[] a;
    private int n;

    public Stiva(int max)
    { a = new int[max]; }

    public void Push(int val)
    { a[n++] = val; }

    public int Pop()
    { return a[--n]; }
}
```

Acesta stivă memorează în câmpul **a** de tip tablou de **int**, un număr de valori întregi. Dacă doriţi să depozitaţi valori de tip **double**, **string** sau oricare alt tip, sunteţi nevoiţi să rescrieţi codul, înlocuind tipul **int** cu noul tip. C# vă oferă posibilitatea să scrieţi doar o singură definiţie de clasă, care să lucreze cu toate tipurile dorite. Veţi scrie o clasă generică.

Clasele generice sunt ***tipuri parametrizate***. Clasa generică se va numi **Stiva<T>** şi se va rescrie astfel:

```
// stiva_generic.cs
using System;

public class Stiva<T>
{
    private T[] a;
    private int n;
    public Stiva(int max)      // Constructor
    { a = new T[max]; }
    public void Push(T val)
    { a[n++] = val; }
    public T Pop()
    { return a[--n]; }
}
```

```
public class TestGeneric
{
    static void Main()
    {
        // Se creează o stivă de int  (T = int)
        Stiva<int> s1 = new Stiva<int>(100);
        s1.Push(2);
        s1.Push(4);
        s1.Push(6);
        Console.WriteLine(s1.Pop() + " " + s1.Pop() + " " +
                          s1.Pop());
        // Se creează o stivă de string (T = string)
        Stiva<string> s2 = new Stiva<string>(50);
        s2.Push("Marcel");
        s2.Push("Ionel");
        s2.Push("Alin");
        Console.WriteLine(s2.Pop() + " " + s2.Pop() + " " +
                          s2.Pop());
    }
}
```

Ieşire:

```
6 4 2
Alin Ionel Marcel
```

Numele clasei generice este `Stiva<T>`. De fapt, nu este o clasă reală, ci este un şablon de clasă. Pe baza şablonului de clasă se vor genera clase reale, pentru valori particulare ale parametrului `T`.

Parametrul şablonului este `T`. Se numeşte ***parametru tip*** sau ***parametru generic***.

Expresia `Stiva<int> s1 = new Stiva<int>(100);` creează pe baza şablonului `Stiva<T>` o clasă cu numele `Stiva<int>`, apoi instanţiază un obiect de acest tip, deci o stivă de întregi.

Argumentele cu care se înlocuiesc ***parametrii tip*** se numesc ***argumente tip***. Mai sus, `T` este *parametru tip*, iar `int` este *argumentul tip*.

O clasă poate avea mai mulţi parametri generici. Programul următor declară o clasa generică (nu uitaţi, o clasă generică este un şablon de clasă), cu doi *parametri tip*, `T1` şi `T2`:

```
using System;

public class A<T1, T2>
{
    private T1 a;
    private T2 b;
    public A(T1 a, T2 b)  // Constructor
    {
        this.a = a;
        this.b = b;
    }
```

```
    public void Print()
    {
        Console.WriteLine(a + " " + b);
    }
}

public class TestGeneric
{
    static void Main()
    {
        A<string, int> m1 = new A<string, int>("Alin", 18);
        m1.Print();
        A<string, string> m2 = new A<string, string>("UNU",
                                                      "DOI");
        m2.Print();
        A<double, char> m3 = new A<double, char>(2.3, 'F');
        m3.Print();
    }
}
```

Ieşire:

```
Alin 18
1 2
2.3 F
```

Expresia `A<string, int> m1 = new A<string, int>("Alin", 18);` generează pe baza şablonului `A<T1, T2>` o clasă cu numele `A<string, int>`, apoi instanţiază un obiect `m1` de acest tip. Similar se generează şi celelalte clase, respectiv obiecte. În felul acesta, metoda `Print()` a primit o funcţionalitate extinsă.

Metode generice

Toate metodele unei clase generice sunt la rândul lor generice, deoarece pot utiliza parametrii generici ai clasei. În afara genericităţii implicite, puteţi defini metode cu proprii parametri generici, care nu depind de cei ai clasei.

Exemplu:

```
using System;

public class C<U>
{
    private U u;

    public C(U u)       // Constructor generic
    {
```

```
        this.u = u;
    }
    // Metoda generică - are proprii parametri generici
    public void F<V1, V2>(V1 a, V2 b)
    {
        Console.WriteLine(a.ToString() + " " + u.ToString() +
                          " " + b.ToString());
    }
}

public class TestMetodaGenerica
{
    static void Main()
    {
        C<int> c1 = new C<int>(104);    // U = int
        c1.F<string, string>("Sa traiti", "ani!");

        C<char> c2 = new C<char>('F');  // U = char
        c2.F<string, int>("Iulia", 8);
    }
}
```

Ieşire:

```
Să traiti 104 ani!
Iulia F 8
```

Expresia `C<int> c1 = new C<int>(104);` generează clasa `C<int>` pe baza şablonului de clasă `C<T>`.
Expresia `c1.F<string, string>("Sa traiti", "ani!");` generează metoda `F<string, string>()` pe baza şablonului de metodă `F<V1, V2>(V1 a, V2 b)`, apoi apelează metoda cu două argumente de tip `string`. Metodele cu proprii parametri generici îşi sporesc funcţionalitatea în raport cu celelalate metode ale unei clase şablon.

Avantajele programării generice

Acest model de programare permite implementarea algoritmilor generici. Pentru asemenea algoritmi, datele se manipulează în acelaşi fel ca în cazul algorimilor non-generici, în timp ce tipurile de date utilizate pot să fie schimbate de către programator după necesităţi. Genericele se remarcă prin calitatea şi eleganţa codului, sintaxa uşor de înţeles. Codul se poate reduce semnificativ ca volum atunci când aveţi sarcini de programare cu cod repetabil.

Folosirea tipurilor generice este recomandată de asemenea pentru siguranţa tipurilor care se creează (corectitudinea tipurilor se verifică în timpul compilării), pentru performanţa în timpul rulării şi nu în ultimul rând pentru creşterea productivităţii în programare.

Colecţii

Platforma **.Net** conţine clase specializate pentru depozitarea datelor. Aceste clase implementează stive, cozi, liste, tabele de dispersie (*hash-tables*). Colecţiile non-generice sunt definite în spaţiul de nume `System.Collections`. Colecţiile generice se definesc în `System.Collections.Generic`.

Cele două tipuri de colecţii implementează aproximativ aceleaşi tipuri de structuri de date, însă cele generice sunt mai performante şi furnizează o mai mare siguranţă a tipurilor. Prezentăm modul de utilizare a câtorva containere generice.

Clasa generică `Stack<T>`

`Stack<T>` este o colecţie de instanţe de acelaşi tip `T`, care implementează operaţii specifice unei stive (*LIFO*). Managementul memoriei se face automat.
Exemplu:

```
using System;
using System.Collections.Generic;

class StivaGenerica
{
    public static void Main()
    {
        // Declară o stivă vidă cu elemente de tip string
        Stack<string> st = new Stack<string>();

        // Adaugă câteva elemente în stivă
        st.Push("UNU"); st.Push("DOI");
        st.Push("TREI"); st.Push("PATRU");
        Console.WriteLine("Nr. de elemente: {0}", st.Count);

        foreach (string s in st)  // Se parcurge stiva
            Console.WriteLine(s);

        // Pop() scoate elementul din vârful stivei
        Console.WriteLine("Scoate '{0}'", st.Pop());

        // Peek() returnează elementul din vârful stivei
        // fără să-l scoată din stivă
        Console.WriteLine("Varful stivei: {0}", st.Peek());
        st.Clear(); // Şterge toate elementele
    }
}
```

Ieşire:

```
Nr. de elemente: 4
PATRU
TREI
```

```
DOI
UNU
Scoate 'PATRU'
Varful stivei: 3
```

Clasa generică List<T>

Clasa reprezintă o listă de obiecte care poate fi accesată prin index. Implementează metode pentru căutarea, sortarea şi manipularea obiectelor. Este echivalentul clasei non-generice **ArrayList**, dar mai performantă decât aceasta.
Exemplu:

```
using System;
using System.Collections.Generic;

public class ListaGenerica
{
    public static void Main()
    {
        // Creează lista copii cu elemente de tip string
        List<string> copii = new List<string>();

        // Adaugă elemente în listă
        copii.Add("Ionel");   copii.Add("Radu");
        copii.Add("Viorel");  copii.Add("Adisor");
        copii.Add("Nelutu");

        // Parcurge colecţia
        foreach (string c in copii)
            Console.Write(c + " " );
        Console.WriteLine("\nNr. copii: {0}", copii.Count);

        // Contains() returnează true dacă un element există
        Console.WriteLine("Viorel exista ? {0}\n",
                                copii.Contains("Viorel"));

        Console.WriteLine("Insereaza \"Alin\" " +
                               "pe pozitia 2:");
        copii.Insert(2, "Alin");

        foreach (string c in copii)
            Console.Write(c + " ");
        Console.WriteLine("\ncopii[3] = {0}", copii[3]);

        Console.WriteLine("\nSorteaza alfabetic: ");
        copii.Sort();
        for (int i = 0; i < copii.Count; i++ )
            Console.Write(copii[i] + " ");
```

```
        Console.WriteLine("\n\nSterge \"Nelutu\":");
        copii.Remove("Nelutu");

        foreach (string c in copii)
            Console.Write(c + " ");

        Console.WriteLine("\n\nSterge lista de copii:");
        copii.Clear();
        Console.WriteLine("Nr. copii: {0}", copii.Count);
    }
}
```

Ieşire:

```
Ionel Radu Viorel Adisor Nelutu
Nr. copii: 5
Viorel exista ? true

Insereaza "Alin" pe pozitia 2:
Ionel Radul Alin Viorel Adisor Nelutu
copii[3] = Viorel

Sorteaza alfabetic:
Adisor Alin Ionel Nelutu Radu Viorel

Sterge "Nelutu":
Adisor Alin Ionel Radu Viorel

Sterge lista de copii:
Nr. de copi: 0
```

Clasa generică `Dictionary<Tkey, Tvalue>`

Clasa este echivalentul clasei **map** din **STL**, **C++**. Realizează o mapare, o corespondenţă biunivocă între o mulţime de chei şi o mulţime de valori. Cheile sunt unice, în sensul că nu pot exista mai multe chei identice în dicţionar. Fiecărei chei îi corespunde o singură valoare asociată.

Clasa oferă metode care implementează operaţii rapide de inserare a perechilor cheie-valoare, de ştergere şi de căutare a valorii după cheia asociată. În dicţionar, perechile cheie-valoare sunt încapsulate în obiecte de tip **KeyValuePair**.

Exemplu:

```
using System;
using System.Collections.Generic;
```

```
public class AgendaTelefonica
{
    public static void Main()
    {
        // Un dicţionar cu cheia string şi valoarea int
        Dictionary<string, int> T =
                          new Dictionary<string, int>();
        // Adaugă elemente in dicţionar.
        T.Add("Ionescu", 209791); // Inserează cu Add()
        T.Add("Pop", 232145);
        T["Vlad"]  = 213048;      // Inserează cu operatorul
        T["Cazacu"]= 219465;      // de indexare

        Console.WriteLine("Cheia Vlad are valoarea: {0}",
                          T["Vlad"]);

        // Cu operatorul de indexare poate schimba valoarea
        // asociată unei chei existente
        T["Vlad"] = 215773;
        Console.WriteLine("Cheia Vlad are valoarea: {0}\n",
                          T["Vlad"]);

        // Dacă cheia nu există in dicţionar, se adaugă o
        // nouă pereche cheie-valoare
        T["Dragnea"] = 279950;

        // ContainsKey() se foloseşte pentru a testa
        // existenţa unei chei înainte de inserare
        // if (!T.ContainsKey("Simion"))
               T.Add("Simion", 200371);

        // In dicţionar elementele se memorează ca perechi
        // cheie-valoare în obiecte de tip KeyValuePair
        foreach (KeyValuePair<string, int> p in T)
            Console.WriteLine("{0, -7}  {1, 10}",
                                    p.Key, p.Value);

        // Şterge o pereche cheie-valoare
        T.Remove("Ionescu");
        if ( !T.ContainsKey("Ionescu") )
            Console.WriteLine("\nCheia Ionescu nu exista");
    }
}
```

Ieşire:

```
Cheia Vlad are valoarea: 213048
Cheia Vlad are valoarea: 215773
```

```
Ionescu    209791
Pop        232145
Vlad       215773
Cazacu     219465
Dragnea    279950
Simion     200371

Cheia Ionescu nu exista
```

Tratarea excepţiilor

În timpul execuţiei unui program pot apărea situaţii excepţionale, cum ar fi operaţii ilegale executate de propriul cod, care pot duce la întreruperea execuţiei programului sau la un comportament neaşteptat. Aceste situaţii se numesc excepţii. C# oferă un mecanism de tratare a excepţiilor, bazat pe cuvintele cheie **try**, **catch** şi **finally**.

Erorile din timpul rulării programului se propagă în program cu ajutorul acestui mecanism de tratare a excepţiilor. Veţi include codul care este probabil să arunce excepţii, într-un bloc **try**. Când excepţia se produce, fluxul de execuţie al programului este dirijat direct în blocul de cod numit **catch**, care "prinde" şi tratează excepţia. Excepţiile "neprinse", sunt captate de către un *handler* furnizat de către sistem, care afişează un mesaj de eroare.

Tipurile exepcepţiilor care pot să apară sunt reprezentate de către clase specializate ale platformei .NET, clase care derivă din clasa **Exception**.

Pentru tratarea excepţiilor, veţi proceda astfel:

```
try
{
    // Cod care poate arunca excepţii
}
catch(System.Execption e)
{
    // Cod care tratează excepţia
}
finally
{
    // Cod care se execută după try-catch indiferent
    // dacă se aruncă sau nu exeсepţii
}
```

Exemplu:

```
using System;

class TestExceptii
```

```
{
    public static void Main()
    {
        int[] a = { 2, 4, 6, 8 };
        try
        {
            Console.WriteLine(a[4]);
        }
        catch (Exception e)
        {
            Console.WriteLine("Exceptie!\n" + e.ToString());
        }
        finally
        {
            Console.WriteLine(a[3]);
        }
    }
}
```

Ieşire:

```
Exceptie!
System.IndexOutOfRangeException: Index was outside the bounds
of the array at TestExceptii.Main() in c:\teste\Program.cs:
line 7
8
```

Blocul **catch** declară o variabilă de tip execpţie (e) care poate fi utilizată pentru a obţine informaţii suplimentare. Codul din blocul **finally** se execută indiferent dacă se aruncă sau nu o excepţie în blocul **try**, permiţând programului să elibereze resursele (fişiere deschise, memorie alocată, etc.). Dacă excepţia se produce, atunci blocul **finally** se execută după **catch**.

Blocul finally poate să lipsească. Pentru acelaşi bloc try, se pot declara mai multe blocuri catch, fiecare dintre ele, precizând o posibilă excepţie care se poate lansa din **try**.

Manevrarea stringurilor

Tipul **string** în C# este un *alias* pentru clasa **System.String** din *.Net Framework*. Este un tip referinţă. Obiectele de tip **string** încapsulează un şir de caractere în format *Unicode*. Stringurile sunt "imutabile". Odată creat, un obiect de tip **string** nu mai poate fi schimbat; toate operaţiile care modifică un string returnează un alt string modificat.

Operaţii şi metode

`string` defineşte operatorii relaţionali =, != şi operatorii de concatenare +, +=. Defineşte de asemenea mai multe metode utile. Vom exemplifica utilizarea câtorva dintre ele.

```
using System;

class TestString
{
    public static void Main()
    {
        string s1 = "Salut", s2 = "salut";
        if (s1 != s2) // Compară stringurile, nu obiectele
            Console.WriteLine("s1 != s2");
        if ( s1.ToUpper() == s2.ToUpper() )
            Console.WriteLine(s1.ToUpper()); //Afişează SALUT

        s1 += s2;   // s1 este acum  "Salutsalut"
        // Inserează şirul " " începand cu poziţia 5
        string s = s1.Insert(5, " ");  // s1 nu se modifică!
        Console.WriteLine(s); // Afişează: "Salut salut"

        // Extrage din s, începând cu pozitia 6,
        // un substring format din 3 caractere
        s = s.Substring(6, 3);
        Console.WriteLine(s);  // Afişează: "Salut salut"
    }
}
```

Formatarea stringurilor

Pentru formatarea stringurilor există în clasa `String` metodele statice `Format()`, supraîncărcate.

Exemplu:

```
using System;
class TestString
{
    public static void Main()
    {
        int x = 123; double y = 23.4589;
        string s = String.Format("x = {0}, y = {1}", x, y);
        Console.WriteLine(s); // Afişează: 123 23.4589
    }
}
```

Mai sus, `{0}` se referă la primul obiect (`x`) din lista de parametri, iar `{1}` identifică cel de-al doilea obiect (`y`).

Opțiunile de formatare sunt diverse. Puteți stabili aliniere *justify* la stânga sau la dreapta şi formatul de afişare a numerelor:

Exemplu:

```
float x = 123.3456F; double y = 23.4589; string s;
s = String.Format("x = {0,12:E3}\ny = {1,12:F2}", x, y);
Console.WriteLine(s);

Afişează:
x =    1.233E+002
y =         23.46
```

Specificatorul `{0,12:E3}` se interpretează astfel: `0` – identifică primul obiect (`x`), `12` este lățimea câmpului de afişare, `E` cere afişare în format ştiințific, iar `3` este numărul de zecimale care se afişează.

Specificatorul `{1,12:F2}` se interpretează astfel: `1` – identifică al doilea obiect (`y`), `12` este lățimea câmpului de afişare, `F` impune afişare în virgulă fixă, iar `2` este numărul de zecimale care se afişează. Dacă doriți aliniere la stânga, se pune semnul `-` după virgulă: `{1,-12:F2}`.

Stringurile se pot formata pentru afişare după aceleaşi reguli şi cu ajutorul metodelor `Console.Write()` şi `Console.WriteLine()`:

Exemplu:

```
double x = 23.4589;
Console.WriteLine("|x = {0,10:F2}|", x);
Console.WriteLine("|x = {0,-10:F2}|", x);
```

Afişează:

```
|x =      23.46|
|x = 23.46     |
```

Transformarea stringurilor în valori numerice

În C# nu există metode sau operatori care să citească date numerice din *stream*-uri şi să le formateze direct în valori numerice, aşa cum sunt funcțiile `scanf()` din limbajul C, sau operatoul de extracție >> din C++. De regulă, veți citi datele ca stringuri. Din aceste stringuri, veți extrage valorile numerice.

Tipurile predefinite (`int`, `double`, `float`, etc.) definesc metoda `Parse()`. Aceasta preia stringul citit şi îl transformă în valoarea numerică corespunzătoare.

Exemplu :

```
int n = int.Parse("652");              // n = 652
double d = double.Parse("-20.235"); // d = -20.235
```

```
// Citim un număr real de la tastatură:
string s = Console.ReadLine();        // s = "91.045"
double f = double.Parse(s);           // f = 91.045
```

Citirea unui şir de valori numerice

Dacă este nevoie să citiţi un şir de valori numerice dintr-un *stream*, de exemplu de la tastatură, atunci trebuie să precizaţi caracterul sau caracterele separatoare între numere.

Un caz simplu este acela în care toate numerele sunt despărţite printr-un singur spaţiu şi se găsesc pe o singură linie. Se citeşte linia într-un string, apoi se desparte stringul în substringuri reprezentând numerele. Pentru aceasta, utilizaţi metoda **Split()**. Aceasta returnează un tablou de stringuri reprezentând numerele citite.

Exemplu:

```
string linie = Console.ReadLine();
string[] s = linie.Split(' ');  // Separatorul este
int x;                          // caracterul spaţiu
foreach (string nr in s)      // Parcurge tabloul de stringuri
{
    x = int.Parse(nr);        // Le transformă în valori int
    Console.Write(x + " ");
}
```

Metoda **Split()** este supraîncărcată. Una dintre versiuni are ca parametru un şir de caractere **char[]**. În acest şir veţi introduce toate caracterele separatoare pe care le consideraţi necesare. Al doilea parametru al metodei, vă permite să înlăturaţi toate substringurile vide din tabloul de stringuri rezultat în urma splitării. Se obţin stringuri vide în situaţia când în stringul iniţial apar doi sau mai mulţi separatori consecutivi.

Exemplu:

```
using System;
public class TestSplit
{
   static void Main()
   {
      char[] sep = { '\n', ' ', '.', '!' };
      string s = "12   27\n...  496!";
      string[] w = null;
      // Tabloul w preia substringurile fără separatori din s
      w = s.Split(sep,StringSplitOptions.RemoveEmptyEntries);

      for (int i = 0; i < w.Length; i++)
      {
         int x = int.Parse(w[i]);
```

```
            Console.Write(w[i] + "_");
         }
      }
}
```

Ieşire:
12_27_496_

Cunoscătorii limbajului C au remarcat desigur similitudinea între metoda **Split()** şi funcţia **strtok()**.

Citiri şi afişări din fişiere de tip text

Toate intrările şi ieşirile în C#, la fel ca în C++ sau Java, presupun folosirea *stream*-urilor. Un *stream* este o reprezentare abstractă a unui dispozitiv fizic (memorie, fişier, reţea, etc.), în care informaţia se poate accesa doar câte un *byte* odată. Într-un stream informaţia circulă într-un singur sens. De exemplu, un program nu va scrie datele direct într-un fişier, ci le va scrie într-un *stream* de ieşire care reprezintă fişierul.

Există *stream*-uri de intrare şi *stream*-uri de ieşire. Din *stream*-urile de intrare programul citeşte date, iar în cele de ieşire scrie date. Păstrând dispozitivul în formă abstractă, destinaţia sau sursa *stream*-ului poate fi ascunsă. Aceasta permite reutilizarea codului, iar codul este similar atunci când aplicaţiile scriu sau citesc din fişiere sau din reţea, de pe disc, din memorie, sau din oricare alt dispozitiv.

.NET Framework defineşte în spaţiul de nume **System.IO** acele clase care reprezintă *stream*-uri cu fişiere. Sunt mai multe astfel de clase. În acest paragraf ne referim la doar un singur aspect: citirea şi scrierea datelor numerice din fişiere text. Iată un program pentru exemplificare:

```
using System;
using System.IO;

class FileReadWrite
{
   static void Main()
   {

      string fin  = @"C:\teste\numere.in";
      string fout = @"C:\teste\numere.out";

      // Deschide fişierele de intrare şi de ieşire
      StreamReader sr = new StreamReader(fin);
      StreamWriter sw = new StreamWriter(fout);

      string[] s = null;
      string linie = null;
```

```
      int x = 0;

      // Citeşte câte o linie până la întâlnirea
      // sfârşitului de fişier
      while ( (linie = sr.ReadLine()) != null )
      {
         s = linie.Split(' '); // Desparte linia in stringuri
         foreach (string nr in s)
         {
            x = int.Parse(nr); // Obţine valoarea numerică
            sw.Write("{0, -5}", x);  // O scrie în fişier
         }
         sw.WriteLine();
      }
      sr.Close();                    // Închide fişierele
      sw.Close();
   }
}
```

Exemplu : Dacă **numere.in** conţine valorile:

```
12 335 561
0 25 5 76
99 1 773 67 3
```

Atunci **numere.out** va avea:

```
12   335  561
0    25   5    76
99   1    773  67   3
```

Observaţii:

- Clasele **StreamReader** şi **StreamWriter** citesc, respectiv scriu caractere din *stream*-uri. Conţin metodele **ReadLine()**, respectiv **WriteLine()**. Acestea citesc sau scriu din *stream* până la caracterul *newline*.
- O constantă şir de caractere se numeşte şir ***verbatim***, dacă se prefaţează cu caracterul @. Exemplu: @"**C:\teste\numere.in**". Efectul este că secvenţele escape din interior nu se mai evaluează. Deci şirul verbatim din exemplu este echivalent cu "**C:\\teste\\numere.in**".
- În expresia **sw.Write("{0, -5}", x);**, 0 semnifică primul obiect de afişat, adică **x**, iar -5 cere alinierea la stânga a rezultatului pe un câmp de lăţime 5.
- Dacă fişierul de intrare poate avea ca separatori între două numere pe aceeaşi linie mai mult decât un singur spaţiu, atunci veţi folosi versiunea **Split()** cu separatori, aşa cum am arătat în paragraful "Transformarea stringurilor în valori numerice".

Rezumatul capitolului

- O delegare este un tip referință, utilizat să încapsuleze o listă ordonată de metode cu aceeaşi semnătură şi acelaşi tip de retur. Lista de metode se numeşte *lista de invocare*. Când un delegat este invocat, el apelează toate metodele din lista de invocare.
- Evenimentele C# permit unei clase sau un obiect să notifice, să înştiințeze alte clase sau obiecte că ceva s-a întâmplat.
- Un eveniment este un membru public al clasei care publică evenimentul. Atunci când are loc o acțiune, acest membru al clasei *publisher* se activează, invocând toate metodele care au subscris evenimentului.
- Scopul programării generice este scrierea de cod care să fie independent de tipurile de date.
- Există clase generice şi metode generice. Clasele generice sunt *tipuri parametrizate*. Parametrii lor se numesc ***parametri tip*** sau ***parametri generici***.
- Colecțiile sunt clase specializate pentru depozitarea datelor. Biblioteca **.NET** defineşte colecții generice şi colecții non-generice.
- Colecțiile implementează stive, cozi, liste, tabele de dispersie (*hash-tables*).
- Tipurile predefinite (`int`, `double`, `float`, etc.) definesc metoda `Parse()`. Aceasta preia *stringul* citit şi îl transformă în valoarea numerică corespunzătoare.
- Clasele `StreamReader` şi `StreamWriter` citesc, respectiv scriu caractere din *stream*-uri. Conțin metodele `ReadLine()`, respectiv `WriteLine()`. Acestea citesc sau scriu din *stream* până la caracterul *newline*.

Întrebări şi exerciții

1. Ce rol îndeplineşte un tip delegat într-un program ?
2. Cum puteți elimina o metodă din lista de invocare a unei delegări ?
3. De ce se spune că aplicațiile Windows sunt *conduse de evenimente* ?
4. Evidențiați asemănările şi deosebirile dintre tipul tablou şi tipul **List<T>**.
5. Să se citească din fişierul text *matrice.in* elementele unui tablou bidimensional de valori întregi. Să se afişeze în fişierul *matrice.out* pătratul valorii fiecărui element al tabloului.

Partea a II - a

Programare Windows cu Visual C# 2008 Express Edition

Capitolul 6

Aplicaţii de tip Windows Forms

Visual C# 2008 Express Edition (**VCSE**) oferă suport pentru dezvoltarea următoarelor tipuri de aplicaţii:

- Aplicaţii Windows de tip *Windows Forms.*
- Aplicaţii Windows de tip **WPF** (*Windows Presentation Foundation*).
- Aplicaţii de tip consolă.
- Aplicaţii de tip bibliotecă dinamică.

Aplicaţiile de tip *Windows Forms* şi cele *WFP* facilitează prin *designer*-ele integrate dezvoltarea interfeţelor grafice cu utilizatorul (*user interface*).

Aplicaţii cu interfaţă grafică cu utilizatorul

Biblioteca .NET conţine un număr mare de clase definite în spaţiul de nume **System.Windows.Forms**, cum sunt: **`Button`**, **`TextBox`**, **`ComboBox`**, **`Label`**, etc. Instanţele acestor clase identifică controalele *Windows.* Controalele sunt elemente de interfaţă grafică ale unei aplicaţii Windows.

Un programator hardcore poate să codeze cu ajutorul acestor clase o aplicaţie cu interfaţă grafică oricât de complicată fără a utiliza mediul integrat, însă munca este imensă. Este de preferat să lăsaţi Visual C# să genereze pentru voi acel cod necesar interfeţei grafice, ca să vă puteţi concentra pe funcţionalitatea aplicaţiei. Veţi utiliza pentru aceasta ***Windows Form Designer*** şi ***Toolbox***. Cu mouse-ul, prin *drag and drop* veţi alege controalele necesare din **Toolbox** şi le veţi aranja pe suprafaţa formelor în **Windows Form Designer**. Veţi seta proprietăţile controalelor. Pe măsură ce faceţi toate acestea, designerul generează codul C# aferent, pe care îl scrie în fişierul **`<nume>.designer.cs`**, unde **`<nume>`** este numele formei respective.

Aşadar, sunt trei etape importante în crearea unei aplicaţii cu interfaţă grafică:

1. **Adăugarea controalelor** pe suprafaţa formelor.
2. **Setarea proprietăţilor iniţiale ale controalelor** din fereastra ***Properties***.
3. **Scrierea handlerelor** pentru evenimente.

Realizarea unei aplicații simple de tip Windows Forms

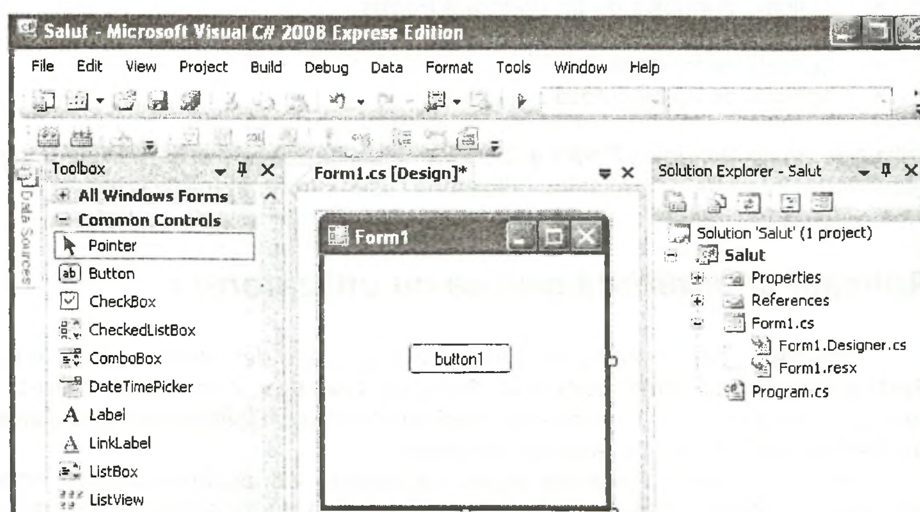
Vom crea o aplicație numită **Salut**, care are un singur buton plasat pe o formă. La apăsarea lui, apare o fereastră sistem de tip `MessageBox`, care afişează un mesaj de salut. Urmați paşii:

1. În meniul **File**, click **New Project**.

2. În fereastra **New Project**, în panoul **Templates**, alegeți ***Windows Forms Application***.

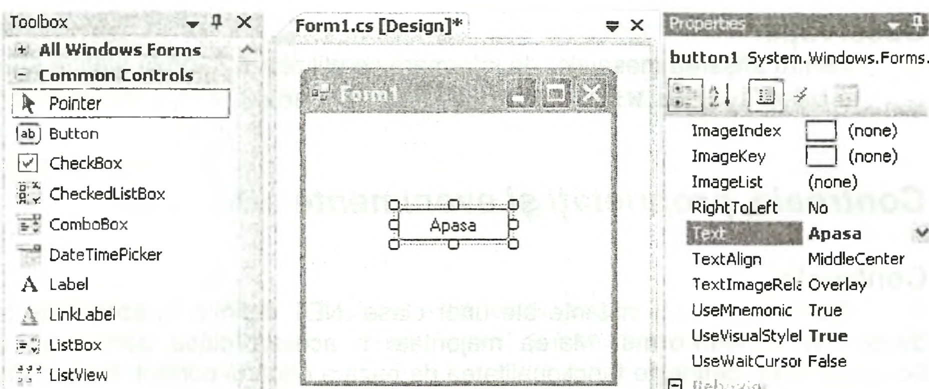
3. În câmpul **Name**, scrieți **Salut**, apoi click **OK**.
   În acest fel ați creat un nou proiect **Windows Forms**. Acum urmează:

4. Din **Toolbox**, trageți un buton pe suprafața formei.



5. Click dreapta pe butonul cu eticheta `button1`. Alegeți *Properties* din meniul contextual.

6. În fereastra **Properties**, schimbați valoarea proprietății **Text** în *Apasa*.



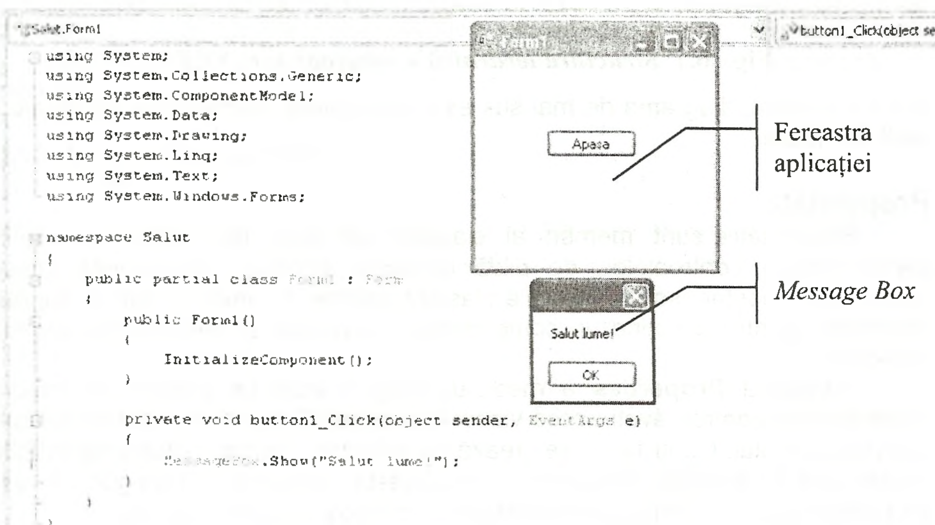
7. Urmează tratarea evenimentului **Click**, pentru ca aplicația să răspundă acestui eveniment. Aveți două variante:
   a. Dublu click pe suprafața butonului.
   b. În fereastra ***Properties***, click pe iconul *"fulger"*, numit *Events*. Din lista de evenimente la care poate răspunde butonul, selectați evenimentul **Click** și apăsați **Enter**.

   În ambele situații se deschide ***Editorul de Cod***. Fișierul deschis este **`Form1.cs`**, iar metoda nou creată este **`button1_Click`**. Acesta este *handler*-ul de eveniment. Va fi invocată în mod automat la click pe buton.

8. În corpul metodei, scrieți codul:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Salut lume!");
}
```

9. Compilați și rulați aplicația cu F5.



Observație:

Pentru afişarea mesajelor de informare se utilizează metoda statică **`Show()`** a clasei **`System.Windows.Forms.MessageBox`**.

Controale, proprietăți şi evenimente

Controale

Controalele sunt instanțe ale unor clase .NET definite în spațiul de nume **System.Windows.Forms**. Marea majoritate a acestor clase derivă din clasa **`Control`**, care defineşte funcționalitatea de bază a oricărui control. Aşa se explică faptul că unele proprietăți şi evenimente sunt comune tuturor controalelor. Clasele de tip control sunt organizate ierarhic, pe baza releției de moştenire, aşa cum se vede din diagrama parțială de mai jos:

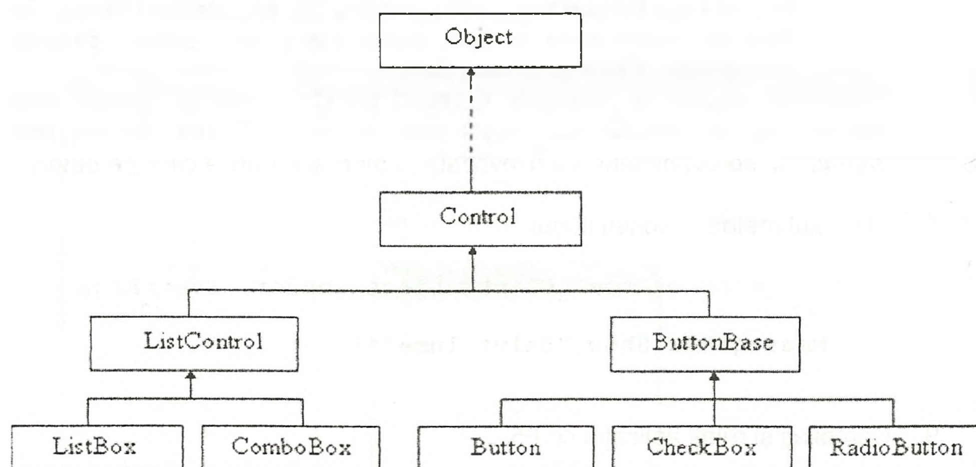


Fig. 6.1 ***Structura ierarhică a controalelor .NET***

În mod evident, diagrama de mai sus este incompletă. Numărul de controale este mult mai mare.

Proprietăți

Proprietățile sunt membri ai claselor din care fac parte. Ele definesc caracteristicile controalelor, de pildă culoarea, poziția, dimensiunile acestora. Controalele moştenesc proprietățile claselor părinte. În unele cazuri le suprascriu (*override*), pentru a obține un comportament particular şi definesc de asemenea altele noi.

Fereastra **Properties** a mediului integrat este un instrument important. Selectând un control, aveți acces vizual la proprietățile şi evenimentele pe care le suportă controlul. Controalele se crează cu adevărat *run-time*, însa proprietățile lor inițiele pot fi stabilite *design-time* în această fereastră. Desigur că aceste proprietăți se pot modifica programatic în timpul execuției programului.

Evenimente

Începând cu această parte a lucrării, ne vom preocupa doar de evenimentele pe care le generează controalele Windows.

IMPORTANT

Programele cu interfață grafică cu utilizatorul sunt ***conduse de evenimente*** *(event-driven)*

În momentul în care utilizatorul acționează asupra unui control, cum ar fi click pe un buton, sistemul de operare "simte" şi transmite controlului un mesaj. Controlul generează atunci un eveniment specific acelei acțiuni, ca un semn că ceva s-a întâmplat. Programatorul poate să trateze sau nu acel eveniment. Dacă alege să o facă, atunci el trebuie să scrie o metodă *handler*, aşa cum este `button1_Click()` în paragraful anterior. Această metodă se apelează în momentul în care evenimentul are loc, iar codul ei asigură funcționalitatea dorită a controlului.

Toate controalele au evenimente pe care le pot genera. Amintiți-vă că evenimentele sunt membrii ai claselor de tip control definite de .NET.

Tratarea evenimentelor

În partea teoretică a lucrării am discutat despre mecanismul tratării evenimentelor în C#. Un obiect publică un eveniment, iar alte obiecte subscriu acestui eveniment. Când evenimentul se declanşează, toate obiectele care au subscris sunt informate, în sensul că *handlerele* acestora se vor invoca.

Să presupunem că avem o formă, pe care ați aşezat un buton. Vrem să tratăm evenimentul click pe buton. Cine este *publisher* ? Desigur, butonul. Şi cine este *subscriber* ? Este fereastra părinte, adică forma. Deci în clasa ataşată formei vom defini o metodă handler. Cum are loc subscrierea ? Vom vedea cu exemplul practic care urmează. Din fericire, codul necesar subscrierii şi definiția metodei de tratare a evenimentului se generează în mod automat, atunci când utilizați panoul **Properties**.

Cum se crează handler-ele

Pentru acelaşi control puteți trata mai multe evenimente, definind desigur câte un *handler* specific. Mai mult decât atat, un acelaşi *handler* poate fi utilizat pentru mai multe controale, aşa cum vom vedea în continuare.

Vom realiza un proiect cu trei controale de tipuri diferite. Pentru fiecare control tratăm evenimentul **Click**. În mod evident, putem scrie câte un handler care să răspundă fiecărui control. Vom proceda însă altfel. Vom crea un singur handler pentru **Click** pe oricare control. Acest lucru este posibil, datorită signaturii speciale a handlerelor .NET:

```
private void NumeHandler(object sender, EventArgs e)
{
      // Cod care tratează evenimentul
}
```

sender este o referință la obiectul (controlul) care a generat evenimentul. Astfel, în corpul handlerului putem identifica acel control și putem trata în mod diferențiat.

Aplicația *ClickEvent*

Pentru realizarea proiectului, urmați pașii de mai jos:

1. Creați un proiect de tip *Windows Forms*, cu numele ***ClickEvent***.

2. Faceți click drept pe suprafața formei și alegeți **Properties**.

3. Schimbați (opțional) titlul formei (implicit este *Form1*) în *ClickEvent*, modificând valoarea proprietății **Text**. Modificați proprietatea **Name** la valoarea *FormaMea*. Este noul nume al clasei formei. Numele implicit era *Form1*.

4. (Opțional). În ***Solution Explorer*** faceți click drept pe fișierul *Form1.cs* și redenumiți-l *FormaMea.cs*. Dacă fereastra ***Solution Explorer*** nu e vizibilă, atunci din meniul **View**, alegeți ***Solution Explorer***.

5. Trageți cu mouse-ul pe suprafața formei din **Toolbox** trei controale diferite, de exemplu un **Button**, un **CheckBox** și un **TextBox**.

6. Selectați butonul și setați proprietatea **Text** la valoarea *Apasare*, iar proprietatea **Name** la valoarea *buton* (implicit aceasta era *button1*).

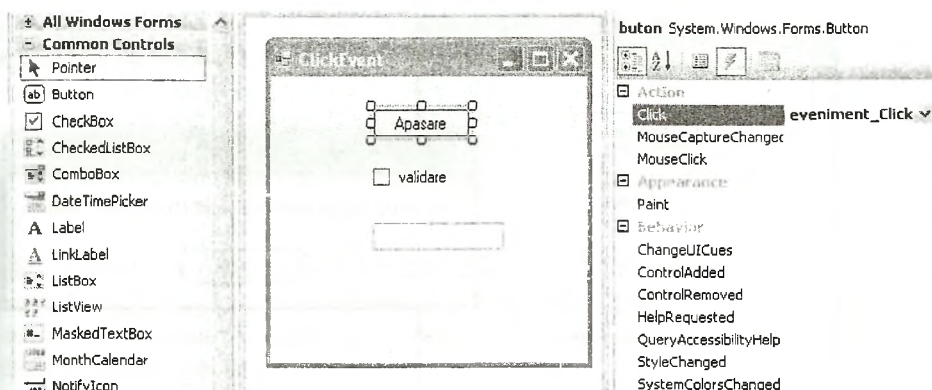
7. Selectați check box-ul și setați proprietatea Text la valoarea *Validare*, iar proprietatea **Name** la valoarea *verif* (implicit aceasta era *checkBox1*).

8. Selectați căsuța de text și atribuiți proprietății **Name** valoarea *edit* (implicit aceasta era *textBox1*).

9. Selectați butonul și din **Properties** apăsați butonul *Events* (fulgerul). Veți crea un *handler* pentru tratarea evenimentului **Click** generat de buton. Aveți varianta simplă de a face dublu click pe eticheta *Click* din **Properties** sau un simplu dublu click pe suprafața butonului. În această variantă se generează metoda cu prototipul:

   ```
   private void buton_Click(object sender, EventArgs e)
   ```

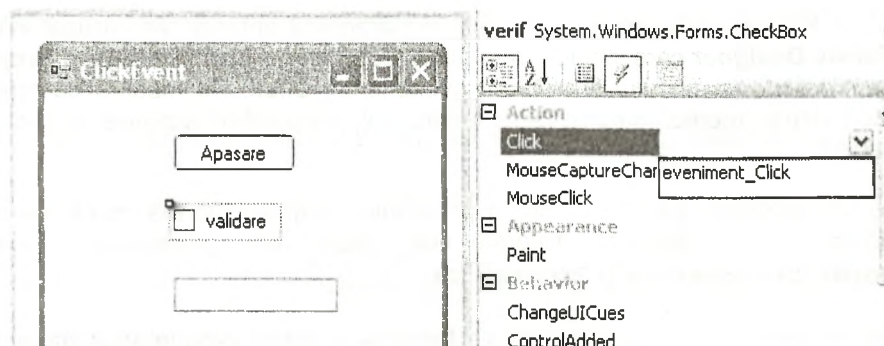
   A doua variantă este să dați alt nume metodei, din fereastra **Properties**, editând câmpul din dreapta etichetei *Click*. Puteți pune de pildă numele **eveniment_Click**, apoi apăsați **Enter**.



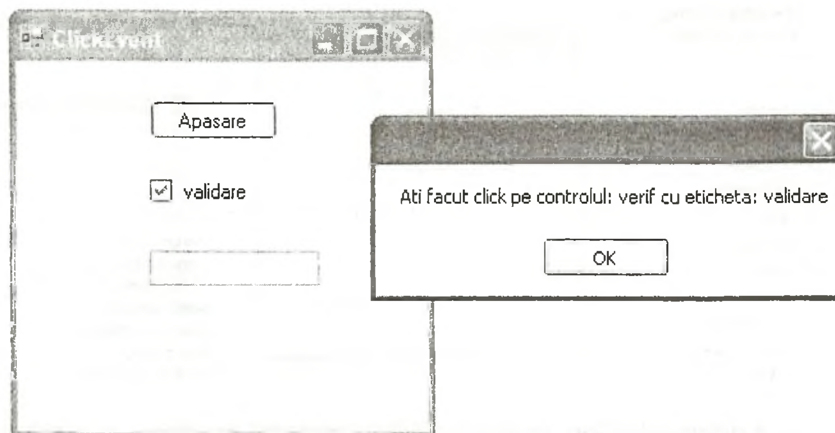
10. În **Editorul de Cod**, completați corpul metodei generate, ca mai jos:

```
private void eveniment_Click(object sender,EventArgs e)
{
   Control c = (Control)sender;  // Conversie explicită
                                 //  (downcast)
   MessageBox.Show("Ati facut click pe controlul: " +
                    c.Name + " cu eticheta: " + c.Text);
}
```

11. Acum subscriem cu acelaşi *handler*, pentru evenimentele generate de **CheckBox** şi **TextBox**. Selectați pe rând câte unul dintre controale, şi din fereastra **Properties**, apăsați butonul *Events*, apoi alegeți din lista din dreapta etichetei *Click* acelaşi *handler*: `eveniment_Click`.



12. Compilați şi rulați cu **F5** (sau click pe butonul cu iconul ▶ din bara de instrumente Standard).



Observaţii:

- În corpul handlerului, `c.Name` şi `c.Text` sunt proprietăţi ale controalelor, reprezentând numele obiectului care reprezintă controlul, respectiv eticheta afişată pe control. **TextBox** nu are proprietatea `Text`.
- Expresia `Control c = (Control)sender;` necesită o conversie explicită (*downcast*), deoarece `sender` e de tip `object`, `Control` e subclasă pentru `Object` (sau `object`), iar conversiile implicite au loc numai de la subclase la superclase.

O privire în spatele scenei

Ne referim la proiectul realizat în paragraful anterior. Am utilizat ***Windows Forms Designer*** care include între altele ***Editorul de Cod*** şi fereastra ***Properties***, pentru scrierea codului şi tratarea evenimentul **Click**. Pe măsură ce operam în mod vizual, mediul integrat lucra pentru noi, translatând acţiunile în cod C#. Să vedem:

A). După primul pas de creare a proiectului, aveam o formă goală, cu eticheta *Form1*. S-au generat fişierele de bază ale proiectului: `Form1.cs`, `Form1.Designer.cs` şi `Program.cs`.

B). În urma plasării controalelor pe formă şi a setării proprietăţilor `Name` şi `Text` pentru formă, buton, butonul de validare şi controlul de tip `TextBox`, dar şi a redenumirii fişierelor proiectului, în ***Solution Explorer*** puteţi identifica: `FormaMea.cs`, `FormaMea.Designer.cs`, `Program.cs`.

Fişierul *Program.cs*

Fişierul conţine metoda `Main()`. Este punctul de intrare în aplicaţie. În **Solution Explorer**, faceţi click drept pe `Program.cs` şi alegeţi *View Code*:

```
// fragment din codul Program.cs
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new FormaMea());
}
```

Partea importantă este faptul că metoda `Main()` din clasa `Program` apelează metoda `Run()`. `Run()` e metodă statică a clasei `Application`. `Run()` lansează în execuție aplicația. O instanță a clasei `FormaMea` este creată şi este facută vizibilă. În acest fişier ar trebui să nu modificați nimic.

Fişierul *FormaMea.cs*

Fişierul implementează în general constructorii clasei, şi toate metodele definite de programator.

În **Solution Explorer**, faceți click drept pe fişierul `FormaMea.cs` şi alegeți *View Code*:

```
// fragment din codul FormaMea.cs
namespace ClickEvent
{
    public partial class FormaMea : Form
    {
        public FormaMea()
        {
            InitializeComponent();
        }
    }

    private void eveniment_Click(object sender, EventArgs e)
    {
       Control c = (Control)sender;  // Conversie explicită
                                     // (downcast)
       MessageBox.Show("Ati facut click pe controlul: " +
                       c.Name + " cu eticheta: " + c.Text);
    }
}
```

Observații:

- Fişerul defineşte clasa `FormaMea`. Cuvântul cheie `partial` spune faptul că definiția clasei se va completa în alt fişier (`FormaMea.Designer.cs`).
- `FormaMea` moşteneşte clasa `Form` din **.NET**, pentru ca să aibă toate caracteristicile unei forme Windows.
- Constructorul clasei, care se invocă la crearea formei, apelează metoda `InitializeComponent()`. Această metodă execută tot codul necesar la

inițializarea aplicației: crearea controalelor, setarea proprietăților inițiale, etc. Cu alte cuvinte, tot ce ați setat *design-time*.
- Practic, dumneavoastră veți coda mai mult în acest fişier. Este locul unde definiți de regulă metode, proprietăți şi evenimente noi ale clasei.
- Oricând doriți ca o metodă să fie apelată odată cu crearea formei, puteți alegeți constructorul clasei în acest scop:

```
public FormaMea()     // Constructor
{
    // Aici puteți apela metode.
    // Controalele nu există încă
    InitializeComponent();
    // Si aici puteți apela metode.
    // Forma şi controalele sunt deja create
}
```

- Când doriți ca un cod să fie executat ca răspuns la o acțiune a user-ului, veți identifica evenimentul care se declanşează la aceaa acțiune, şi-l veți trata. Veți scrie codul dorit în *handler*-ul de eveniment.

Fişierul *FormaMea.Designer.cs*

Fişierul reține tot ce s-a stabilit în etapa de *design*, relativ la formă şi controale. Codul fişierului se generează în mod automat.

În ***Solution Explorer***, faceți click drept pe fişierul **FormaMea.Designer.cs** şi alegeți *View Code*:

```
// fragment din codul FormaMea.Designer.cs
namespace ClickEvent
{
    partial class FormaMea
    {
        // ...
        private void InitializeComponent()
        {
            // ...
        }
        private System.Windows.Forms.Button buton;
        private System.Windows.Forms.CheckBox verif;
        private System.Windows.Forms.TextBox edit;
    }
}
```

- **IMPORTANT!** Fişerul completează definiția clasei **FormaMea** reținând toate setările facute *design-time*.
- **IMPORTANT!** Remarcăm trei câmpuri private: referințele **buton**, **verif** şi **edit**. Câmpurile au apărut sub numele: **button1**, **checkBox1** şi

textBox1, în momentul în care aţi plasat controalele pe formă. Numele actuale s-au generat automat, atunci când din fereastra **Properties** aţi modificat proprietatea **Name**.

- **IMPORTANT!** Cele trei referinţe sunt **conţinute** în clasa `FormaMea`. Amintiţi-vă relaţia care se modelează la ***conţinere***: ***HAS A***. Întradevăr, forma ***are un*** sau ***conţine un*** buton, ***are un*** check box şi ***are un*** text box.

Metoda *InitializeComponent()*

Redăm fragmentele semnificative de cod din acestă metodă:

```
private void InitializeComponent()
{
    // Instanţierea (crearea) celor trei controale
    // prin apelul constructorilor
    this.buton = new System.Windows.Forms.Button();
    this.verif = new System.Windows.Forms.CheckBox();
    this.edit  = new System.Windows.Forms.TextBox();

    // Setarea proprietăţilor iniţiale ale controlului
    // buton ca urmare a acţiunilor de design în fereastra
    // Properties şi Form Designer.
    this.buton.Location = new System.Drawing.Point(78, 23);
    this.buton.Name = "buton";
    this.buton.Size = new System.Drawing.Size(75, 23);
    this.buton.Text = "Apasare";

    // Subscrierea la evenimentul Click
    this.buton.Click
           += new System.EventHandler(this.eveniment_Click);

    // Setări similare pentru celelalte două controale
    // ...

    // Adăugarea controalelor pe formă
    this.Controls.Add(this.edit);
    this.Controls.Add(this.verif);
    this.Controls.Add(this.buton);

    // Setarea proprietăţilor formei
    this.Name = "FormaMea";
    this.Text = "ClickEvent";

    // Afişarea controalelor pe formă
    this.ResumeLayout(false);
    this.PerformLayout();
}
```

Se impun câteva precizări cu privire la metoda `InitalizeComponent()`:

1. Codul metodei se generează în mod automat. **Nu este indicat să modificați manual acest cod**. Operați cu ***Form Designer*** și fereastra **Properties** și modificările se produc de la sine.
2. Metoda construiește controalele prin apelul constructorilor acestora.
3. Metoda stabilește proprietățile inițiale ale controalelor și ale formei.
4. Metoda stabilește subscrierea formei la evenimentele generate de către controale. Subscrierea se face astfel:

```
this.buton.Click += new
             System.EventHandler(this.eveniment_Click);
this.edit.Click += new
             System.EventHandler(this.eveniment_Click);
this.verif.Click += new
             System.EventHandler(this.eveniment_Click);
```

   `this` este referința la obiectul de tip `FormaMea`. O subscriere ca aceasta este la fel de corectă:

```
buton.Click += eveniment_Click;
edit.Click  += eveniment_Click;
verif.Click += eveniment_Click;
```

   Altfel spus, obiectul referit de `this` (forma) subscrie la evenimentele **Click** generate de către fiecare control, setând pentru fiecare caz, același *handler*: `eveniment_Click`.
5. Puteți, cu siguranță, să tratați un eveniment oarecare, scrind manual codul necesar subscrierii și definind un handler adecvat, fără utilizarea panoului **Properties**. Totuși, e mult mai comod să utilizați facilitățile mediului integrat.
6. Ca idee generală, numele fișierului `<nume_formă>.Designer.cs`, ne spune că este generat în întregime de către compilator. În general, nu se fac modificări manuale în acest fișier.

Declanșarea programatică a unui eveniment

Puteți constrânge programatic un control să genereze evenimentul **Click**, chiar dacă userul nu a acționat click pe acel control. Se utilizează metoda `PerformClick()`. Vom face un mic proiect:

Aplicația *RaiseEvent*

1. În meniul **File**, alegeți **NewProject**.
2. În panoul **Templates**, selectați **Windows Forms Application**.
3. In câmpul **Name**, scrieți de exemplu *RaiseEvent* ca nume de proiect.

4. Din **Toolbox**, aduceți pe formă două butoane.

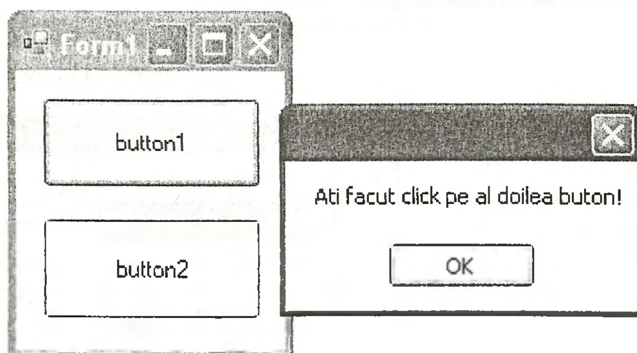
5. Faceți dublu click pe primul buton pentru a trata evenimentul **Click**. În **Editorul de Cod**, scrieți în corpul *handler*-ului:

```
private void button1_Click(object sender, EventArgs e)
{
    // Declanşează evenimentul Click pentru al doilea
    // buton
    button2.PerformClick();
}
```

6. Faceți dublu click pe al doilea buton pentru a trata evenimentul **Click**. În **Editorul de Cod**, scrieți în corpul *handler*-ului:

```
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("Ati facut click butonul doi !");
}
```

7. Apăsați **F5** pentru compilare şi rulare.



Mesajul se afişează indiferent de butonul pe care faceți click.

Crearea programatică a unui control

Până în prezent, controalelelor prezente pe formă le-au fost setate proprietățile *design–time*. Crearea lor are loc totuşi *run-time*, odată cu crearea formei. Puteți utiliza oricare *handler* de eveniment pentru a crea şi a seta proprietăți *run-time* pentru oricare control Windows.

Aplicația *DtpRuntime*

Realizăm o aplicație care în timpul execuției, prin apăsarea unui buton, crează un nou control pe formă, de exemplu un **`DateTimePicker`**. Un control **`DateTimePicker`** afişează data curentă.

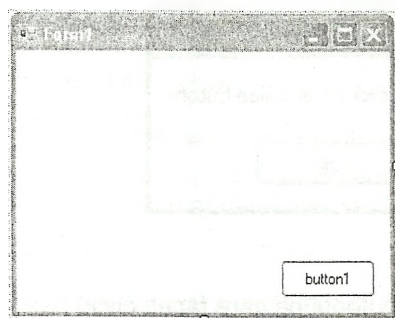
1. În meniul **File**, alegeți **NewProject**.
2. În panoul **Templates**, selectați **Windows Forms Application**.
3. In câmpul **Name**, scrieți de exemplu *DtpRuntime* ca nume de proiect.
4. Din **Toolbox**, trageți pe formă un buton.
5. Faceți dublu click pe suprafața butonului pentru a trata evenimentul **Click**. În **Editorul de Cod**, scrieți în corpul *handler*-ului:

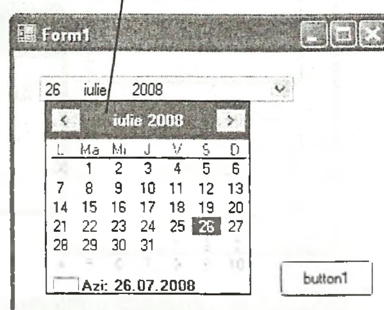
```
private void button1_Click(object sender, EventArgs e)
{
    // Crează runtime un obiect de tip D.T.P.
    DateTimePicker dtp = new DateTimePicker();

    // Location e poziția colțului stânga sus al D.T.P.
    dtp.Location = new Point(20, 20);

    // Adaugă controlul pe formă
    this.Controls.Add(dtp);
}
```

Control creat *runtime* la apăsarea butonului *button1*





Capitolul 7

Controalele Windows Forms

Există o mare diversitate de controale predefinite .NET. Într-o clasificare relativă după funcționalitate, distingem:

- Controale pentru declanşarea evenimentelor, cum este **`Button`**.
- Controale pentru editare de text, cum sunt **`TextBox`** sau **`RichTextBox`**.
- Controale pentru afişare de informații pentru utlizator, cum sunt **`Label`** şi **`LinkLabel`**.
- Controale pentru afişare de liste, cum sunt **`ListView`** sau **`ListBox`**.
- Controale pentru afişarea informațiilor din baze de date, cum este **`DataGridView`**.
- Controale de tip container, cum sunt **`GroupBox`** sau **`Panel`**.
- Controale de tip meniu sau bară de instrumente, cum sunt **`MenuStrip`** sau **`ToolStrip`**.

La cele de mai sus se mai adaugă şi alte tipuri de controale, care vor fi descrise în cele ce urmează, însoțite de exemple practice.

Controlul Button

În spațiul de nume **System.Windows.Forms** se definesc trei controale care derivă din clasa **`ButtonBase`**: **`Button`**, **`CheckBox`** şi **`RadioButton`**.

Button este de departe cel mai comun control. Pentru a inspecta proprietățile şi evenimentele unui buton, este suficient să creați rapid o aplicație de tip *Windows Forms*, să aduceți pe formă un buton şi să parcurgeți fereastra **Properties**. De multe ori însă nu este de ajuns pentru a vă lămuri. De aceea, vă sfătuim ca să aveți mereu deschis *Helpul* mediului integrat şi să vă obişnuiți să căutați rapid acolo oricare clasă, proprietate sau cod exemplificator. Din meniul **Help**, alegeți **Contents**. În panoul din stânga, expandați nodul **.Net Framework SDK**. Apoi expandați **.Net Framework Class Library**. Expandând mai departe nodul **System.Windows.Forms Namespace**, întreaga galerie de clase de tip control vă este accesibilă cu toate informațiile necesare.

Revenind la **Button**, acesta poate genera mai multe tipuri de evenimente, însă în proporție covârşitoare, evenimentul **Click** este cel mai folosit.

Aplicația *ButtonExample*

Vom face o mică aplicație care tratează alte două tipuri de evenimente şi pune în evidență câteva dintre proprietățile butoanelor.

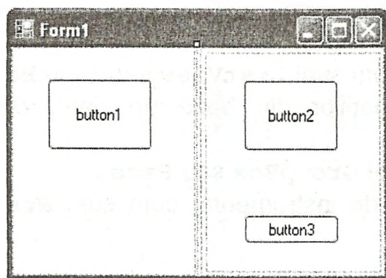
1. În meniul **File**, alegeți **NewProject**.

2. În panoul **Templates**, selectați *Windows Forms Application*.

3. In câmpul **Name**, scrieți de exemplu ***ButtonExample*** ca nume al proiectului.

4. Din **Toolbox**, trageți cu mouse-ul pe suprafața formei un control de tip **SplitContainer**. Acesta are aici rolul de a izola partea din stânga de cea din dreapta a formei.

5. În panoul din stânga aduceți un buton, iar în cel din dreapta, plasați două butoane.



6. Acționați dublu click pe *button1* pentru a trata evenimentul **Click**. În corpul handlerului introduceți codul:

```
private void button1_Click(object sender, EventArgs e)
{
    // Dacă butonul este andocat (umple panoul)
    if (button1.Dock == DockStyle.Fill)
    {
        // Îl readucem la forma şi eticheta iniţiale
        button1.Dock = DockStyle.None;
        button1.Text = "button1";
    }
    else
    {
        // Îl andocăm şi schimbăm eticheta
        button1.Dock = DockStyle.Fill;
        button1.Text = "Fill !!";
    }
}
```

7. Selectați *button2* şi din fereastra **Properties** setați proprietatea **BackColor** la valoarea **Red**. şi proprietatea **Text** la valoarea **Rosu**.

8. Selectați *button2*, iar din fereastra **Properties**, apăsați butonul *Events* (fulgerul). Faceți dublu click pe evenimentul **MouseEnter**. Acesta se declanşează atunci când mouse-ul intră pe suprafața controlului. În corpul *handler*-ului scrieți codul:

```
private void button2_MouseEnter(object sender,
                                          EventArgs e)
{     // Setăm fundalul la culoarea galbenă
      button2.BackColor = Color.Yellow;
      button2.Text = "Galben";
}
```

9. Selectați *button2* și faceți dublu click pe evenimentul **MouseLeave**. Acesta se declanșează atunci când mouse-ul iese de pe suprafața controlului. În corpul *handler*-ului scrieți codul:

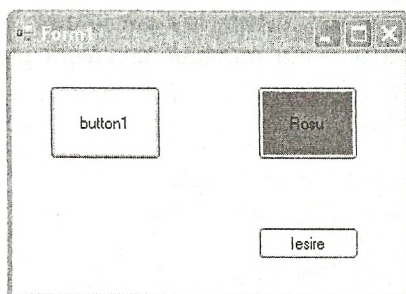
```
private void button2_MouseLeave(object sender,
                                          EventArgs e)
{     // Resetăm culoarea fundalului şi textul
      button2.BackColor = Color.Red;
      button2.Text = "Rosu";
}
```

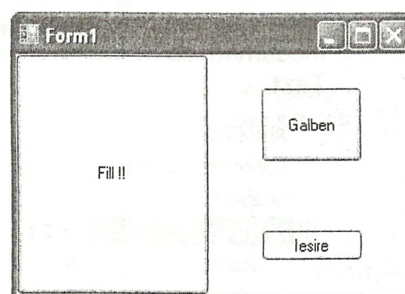
10. Selectați *button3* și din fereastra **Properties** setați proprietatea **Text** la valoarea `Iesire`.

11. Faceți dublu click pe butonul cu eticheta *Iesire*, pentru a trata evenimentul **Click**. Introduceți în corpul handlerului de eveniment, codul:

```
private void button3_Click(object sender, EventArgs e)
{
      Application.Exit(); // Ieşire din aplicaţie
}
```

12. Apăsați pentru rulare **F5**, sau iconul ▸ .





La click pe *button1*, acesta se andochează, apoi la un nou click, revine la starea inițială. La intrarea cu mouse-ul deasupra butonului cu eticheta *Roşu*, acesta își schimbă culoarea și eticheta, iar la ieșirea mouse-ului revine la starea inițială. Utilizați metoda `Application.Exit()` de câte ori doriți să încheiați aplicația.

Controalele Label şi LinkLabel

Controalele `Label` au menirea de a plasa un text pe o formă, cu informaţii pentru utilizator. Aşa cum puteţi vedea din fereastra **Properties**, controalele `Label` suportă evenimente, însă de regulă nu le veţi trata.

.NET Framework defineşte şi clasa `LinkLabel`, derivată din `Label`. Aceasta afişează o porţiune din text ca un *hyperlink*. La un click pe *hyperlink*, puteţi deschide pagini WEB, lansa în execuţie diverse diverse aplicaţii şi în general puteţi utiliza *handler*-ul de tratare a evenimentului **Click**, pentru orice acţiune.

Aplicaţia *LabelExample*

Vom realiza un proiect care exemplifică cel mai simplu mod de utilizare a etichetelor.

1. În meniul **File**, alegeţi **NewProject**.

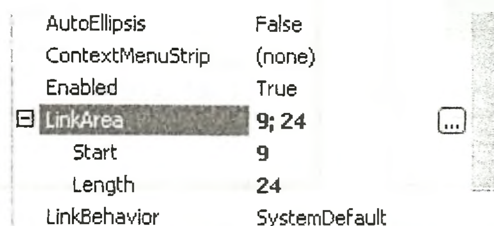
2. În panoul **Templates**, selectaţi **Windows Forms Application**.

3. In câmpul **Name**, scrieţi de exemplu ***LabelExample*** pentru numele proiectului.

4. Din **Toolbox**, trageţi cu mouse-ul pe suprafaţa formei un control de tip `Label`, şi două controale de tip `LinkLabel`.

5. Selectaţi controlul de tip **Label** şi în fereastra ***Properties*** setaţi valoarea proprietăţii **Text** la valoarea `"Un control de tip Label"`.

6. Selectaţi primul control **LinkLabel** şi în fereastra ***Properties*** setaţi proprietatea **Text** la valoarea "*Vizitati www.yahoo.com*", apoi expandaţi nodul `LinkArea` şi setaţi proprietatea **Start** la valoarea **9**. Aceasta înseamnă că link-ul începe după al 9-lea caracter al valorii proprietăţii **Text**.



7. Selectaţi primul control de tip **LinkLabel**, iar în fereastra ***Properties***, apăsaţi butonul *Events*. Faceţi dublu click pe evenimentul **LinkClicked**. Scrieţi în corpul *handler*-ului codul:

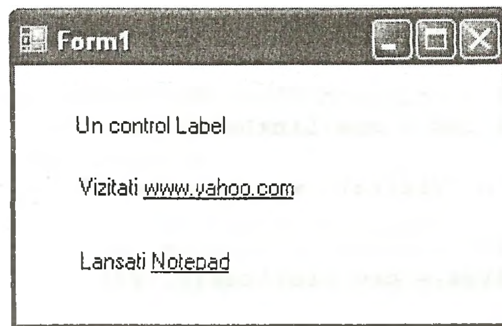
```
private void linkLabel1_LinkClicked(object sender,
                    LinkLabelLinkClickedEventArgs e)
```

```
{
    // Schimbă culoarea link-ului după click
    linkLabel1.LinkVisited = true;

    // Schimbă textul în controlul Label
    label1.Text = "Se viziteaza www.yahoo.com";

    // Foloseste metoda Process.Start() pentru a
    // deschide un URL cu browserul implicit
    System.Diagnostics.Process.Start(
                              "http://www.yahoo.com");
}
```

8. Selectați al doilea control **LinkLabel**. În fereastra ***Properties*** setați proprietatea **Text** la valoarea "*Lansati Notepad*", apoi expandați nodul **LinkArea** și setați proprietatea **Start** la valoarea **8**. Aceasta înseamnă că *link*-ul începe după al 8-lea caracter al valorii proprietății **Text**:

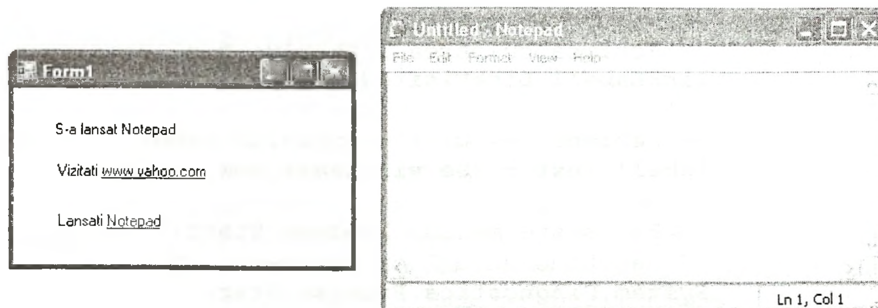


9. Selectați controlul **LinkLabel** cu textul "*Lansati Notepad*", iar în fereastra ***Properties*** apăsați butonul *Events*. Faceți dublu click pe evenimentul **LinkClicked**. Scrieți în corpul *handler*-ului codul:

```
private void linkLabel2_LinkClicked(object sender,
                    LinkLabelLinkClickedEventArgs e)
{
    linkLabel2.LinkVisited = true;
    label1.Text = "S-a lansat Notepad";

    // Folosește metoda Start() pentru a lansa Notepad
    System.Diagnostics.Process.Start("notepad");
}
```

10. Lansați în execuție cu **F5**.



Observații:

- Proprietatea **LinkArea** a unui control **LinkLabel**, este o structură care reține poziția primului caracter al link-ului şi numărul de caractere din link. Iată cum puteți crea programatic un **LinkLabel** care va fi adăugat pe o formă:

```
void AddLinkLabel()
{
    // Crează o etichetă
    LinkLabel lnk = new LinkLabel();

    lnk.Text = "Vizitati www.yahoo.com";

    // 9 -poziția primului caracter al link-ului
    lnk.LinkArea = new LinkArea(9, 24);

    // Poziția de amplasare pe formă
    lnk.Location = new Point(20, 20);

    // Adaugă controlul pe forma curentă
    this.Controls.Add(lnk);
}
```

  Această metodă se poate apela în constructorul clasei formei, sau într-un *handler* de eveniment.

- Metoda **Process.Start()** din spațiul de nume **System.Diagnostic** este supraîncărcată şi este utilă oricând doriți să lansați în execuție o altă aplicație din propria aplicație. De exemplu, apelul următor deschide fişierul *Salut.doc* cu *WordPad*:

```
System.Diagnostics.Process.Start("wordpad.exe",
                                  @"c:\Salut.doc");
```

Controalele RadioButton, CheckBox şi GroupBox

Aşa cum am mai arătat, controalele **`Button`**, **`RadioButton`** şi **`CheckBox`** derivă direct din clasa **`ButtonBase`**.

Un buton radio poate fi apăsat sau nu. Se foloseşte atunci când utilizatorul trebuie să facă o singură alegere între mai multe opţiuni, iar acele opţiuni se exclud reciproc. De exemplu, trebuie să marcheze dacă e bărbat sau femeie, dacă e căsătorit sau nu, etc.

Butoanele radio nu se pun niciodată direct pe formă. Ele se grupează într-un control container, de regulă un **`GroupBox`**. În interiorul unui control container, radio butoanele devin coerente din punct de vedere logic, în sensul că un singur buton poate fi selectat la un moment dat.

Butoanele de validare (*checkbox*-urile) permit utilizatorului să aleagă una sau mai multe opţiuni. De exemplu, utilizatorul poate sa bifeze mai multe sporturi preferate. Pot să fie sau nu incluse în controale container.

Controalele **`RadioButton`** şi **`CheckBox`** au proprietatea numită **`Checked`**, care indică dacă controlul este selectat sau nu.

Aplicaţia *GroupRadioCheck*

Proiectul ilustrează un mod simplu de utilizare a controalelor **`RadioButton`**, **`CheckBox`** şi **`GroupBox`**.

1. Din meniul **File**, alegeţi **NewProject**.

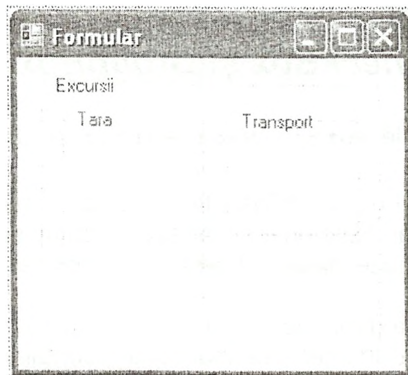
2. În panoul **Templates**, selectaţi **Windows Forms Application**.

3. In câmpul **Name**, scrieţi de exemplu ***GroupRadioCheck*** pentru numele proiectului.

4. Selectaţi forma. În fereastra ***Properties*** setaţi proprietatea **Text** la valoarea *Formular.*

5. Din **Toolbox**, trageţi cu mouse-ul pe suprafaţa formei un control de tip **`GroupBox`**. În fereastra ***Properties***, etichetaţi controlul, setând proprietatea **Text** la valoarea *Excursii.*

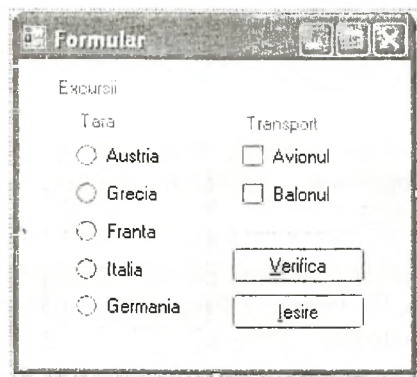
6. Din **Toolbox**, trageţi cu mouse-ul pe suprafaţa primului **GroupBox**, încă două controale de tip **GroupBox**. În fereastra **Properties**, selectaţi-le pe rând şi etichetaţi-le astfel: *Tara*, respectiv *Transport.*



7. Pe suprafața groupbox-ului cu numele *Tara*, aduceți din **Toolbox** cinci radio butoane. Selectați-le pe rând şi din fereastra **Properties** setați-le proprietățile **Text** (etichetele) la valorile: *Austria*, *Grecia*, *Franta*, *Italia*, *Germania*.

8. Pe suprafața groupbox-ului cu numele *Transport*, aduceți din **Toolbox** două butoane de validare. Selectați-le pe rând şi din fereastra **Properties** setați-le proprietățile **Text** (etichetele) la valorile: *Avionul*, respectiv *Balonul*.

9. Pe suprafața groupbox-ului cu eticheta *Excursii*, aduceți două butoane. Selectați primul buton şi setați proprieatea **Text** la valoarea *&Verifica*. Ampersandul are ca efect sublinierea caracterului care îi urmează, informând userul că poate să acceseze butonul de la tastatură cu combinația: *Alt + litera subliniată*. Selectați cel de-al doilea buton şi setați-i proprietatea **Text** la valoarea *&Iesire*.



10. Faceți dublu click pe butonul *Verifica*, pentru a trata evenimentul **Click**. În *handler*-ul de eveniment, scrieți codul evidențiat cu ***bold***:

```
private void button1_Click(object sender,
                                         EventArgs e)
```

```
{
    string msg = "Vom pleca in ";
    // Verifică care dintre butoanele radio e apăsat
    if (radioButton1.Checked)
        msg += radioButton1.Text;
    if (radioButton2.Checked)
        msg += radioButton2.Text;
    if (radioButton3.Checked)
        msg += radioButton3.Text;
    if (radioButton4.Checked)
        msg += radioButton4.Text;
    if (radioButton5.Checked)
        msg += radioButton5.Text;

    // Verifică starea butoanelor de validare
    bool firstDest = false;
    if (checkBox1.Checked)
    {
        msg += " cu " + checkBox1.Text;
        firstDest = true;
    }
    if (checkBox2.Checked)
        if (firstDest)
            msg += " si cu " + checkBox2.Text;
        else
            msg += " cu " + checkBox2.Text;

    // Afişează mesajul
    MessageBox.Show(msg + " !");

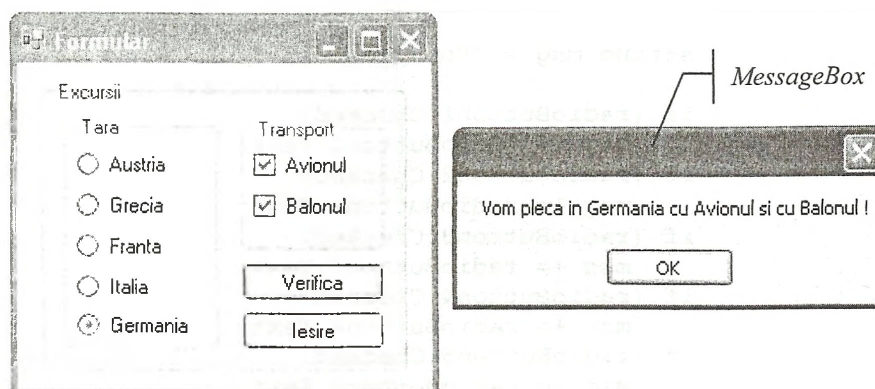
    // Restabileşte starea iniţială
    checkBox1.Checked = false;
    checkBox2.Checked = false;
    radioButton1.Checked = true;
}
```

11. Faceți dublu click pe butonul *Iesire*, pentru tratarea evenimentului **Click**. În *Editorul de Cod* veți scrie:

```
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit(); // Ieşire din aplicaţie
}
```

12. Rulați aplicația cu **F5** sau acționați butonul ▸ din *Standard Toolbar*.

La rulare, obțineți:



Controlul TextBox

Platforma .NET oferă două controale pentru editare de text: **`TextBox`** şi **`RichTextBox`**. Ambele clase derivă din clasa **`TextBoxBase`**. **`TextBox`** este un control cu mai puţine facilităţi decât **`RichTextBox`**, însă este foarte util pentru introducerea de date mici ca dimensiuni de la tastatură.

Principalii membri ai clasei TextBox

De la clasa părinte **`TextBoxBase`**, moşteneşte proprietăţi şi metode pentru manipularea textului, cum este selectarea textului, copiere, tăiere şi lipire din *clipboard*, ca şi un mare număr de evenimente. Descriem o parte dintre membrii clasei.

Proprietăţi :

`Multiline`	-	Returnează sau setează o valoare booleană care indică dacă *text box*-ul poate avea mai multe linii
`AcceptsReturn`	-	Returnează sau setează o valoare care indică dacă la apăsarea **Enter** într-un **TextBox** multilinie se va trece la linie nouă sau se activează butonul implict al formei
`PassworChar`	-	Returnează sau setează un caracter folosit să mascheze parola
`Text`	-	Returnează sau modifică textul în control

Metode:

`Copy()`	-	Copiază textul selectat în *Clipboard*
`Paste()`	-	Lipeşte în control conţinutul *Clipboard*-ului

`Cut()`	-	Mută textul selectat în *Clipboard*
`Clear()`	-	Şterge textul din control
`Select()`	-	Selectează un interval de text în control
`SetBounds()`	-	Setează limitele controlului, la locaţia şi la dimeniunile specificate
`Undo()`	-	Anulează ultima operaţie efectuată în *text box*

Evenimente:

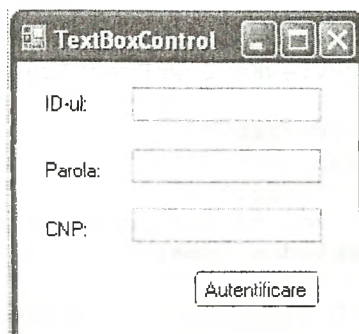
`Click`	-	Se declanşează la click în control
`GotFocus`	-	Se declanşează cînd controlul primeşte focusul
`Leave`	-	Se declanşează cînd focusul părăseşte controlul
`TextChanged`	-	Se declanşează când proprietatea `Text` se schimbă.

Aplicaţia *TextBoxExample*

Proiectul următor relevă câteva dintre caracteristicile controlului **TextBox**.

1. Creaţi un proiect de tip ***Windows Forms Application***, cu numele *TextBoxExample*.

2. Din **Toolbox**, plasaţi cu mouse-ul pe suprafaţa formei trei controale de tip **Label**, trei controale de tip **TextBox** şi un buton, apoi aranjaţi-le aşa ca în figură:



3. Selectaţi primul control de tip **TextBox**. Din fereastra **Properties** setaţi proprietatea **Name** la valoarea **`idTextBox`**. Setaţi în acelaşi mod pentru al doilea control proprietatea **Name** la valoarea **`parolaTextBox`** şi **`cnpTextBox`** pentru al treilea control de tip **`TextBox`**. Pentru eticheta *CNP*, setaţi proprietatea **Name** la valoarea `cnpLabel`, iar pentru buton, **`autentifButton`**.

4. Inițial, numai primul câmp text este activ, celelalte două fiind dezactivate. În acest scop se utilizează proprietatea `Enabled`. Dorim ca încă de la apariția formei controalele `parolaTextBox` și `cnpTextBox` să fie dezactivate. Pentru aceasta, vom scrie codul necesar în constructorul formei. Selectați forma, apoi în **Properties** setați câmpul **Name** la valoarea *TextBoxEx*. În felul acesta, ați modificat numele clasei formei, care inițial era *Form1*. Faceți click drept în ***SolutionExplorer*** pe numele formei și alegeți *View Code*. În constructorul clasei introduceți următoarele:

```
public TextBoxEx()    // Constructorul formei
{
      InitializeComponent();
      // Caracterul '*' maschează parola
      parolaTextBox.PasswordChar = '*';

      // Controlul de tip TextBox pentru parolă
      // este inițial vizibil, dar dezactivat
      parolaTextBox.Enabled = false;

      // Eticheta și TextBox-ul pentru CNP, ca și
      // butonul Autentificare sunt inițial invizibile
      cnpLabel.Visible   = false;
      cnpTextBox.Visible = false;
      autentifButton.Visible = false;
}
```

5. Pentru testarea validității ID-ului introdus, tratați evenimentul **PreviewKeyDown**:

```
private void idTextBox_PreviewKeyDown(object sender,
                          PreviewKeyDownEventArgs e)
{
      // Dacă tasta apăsată este Tab
      if (e.KeyCode == Keys.Tab)
        if (idTextBox.Text == "ionel")
            // ID corect, activăm câmpul pentru parolă
            parolaTextBox.Enabled = true;
        else
            MessageBox.Show("Utilizatorul nu exista!");
}
```

6. Tratați evenimentul **PreviewKeyDown** pentru verificarea parolei:

```
private void parolaTextBox_PreviewKeyDown
      (object sender, PreviewKeyDownEventArgs e)
{
     // Dacă tasta apăsată este Enter
     if (e.KeyCode == Keys.Enter)
```

```
        {
            // Dacă câmpul Text are valoarea "parola"
            if (parolaTextBox.Text == "parola")
            {
                // Eticheta şi controlul de tip text
                // pentru CNP devin vizibile
                cnpLabel.Visible  = true;
                cnpTextBox.Visible = true;

                // TextBox-ul pentru CNP primeşte focusul
                cnpTextBox.Focus();

                // Butonul devine vizibil pe formă
                autentifButton.Visible = true;
             }
             else
             {
                MessageBox.Show("Parola incorecta!");
                // Şterge textul din control
                parolaTextBox.Clear();
             }
        }
    }
```

7. Pentru CNP, dorim să nu permitem introducerea altor caractere decât cifre. Pentru aceasta, trataţi evenimentul **KeyPress** şi pentru toate caracterele cu codul *ASCII* mai mic decât **48** sau mai mare decât **57**, setaţi proprietatea `Handled` la valoarea `true`. În felul acesta, evenimentul nu mai este transmis controlului, iar caracterul apăsat nu va mai fi afişat.

```
private void cnpTextBox_KeyPress(object sender,
                                    KeyPressEventArgs e)
{     // Daca nu e cifră, nu se afişează
      if (e.KeyChar < 48 || e.KeyChar > 57)
          e.Handled = true;
}
```

8. Trataţi evenimentul **Click** pentru butonul *Autentificare*. Dublu click pe buton în *Form Designer*. Veţi cere ca numărul total de cifre să fie **13**:

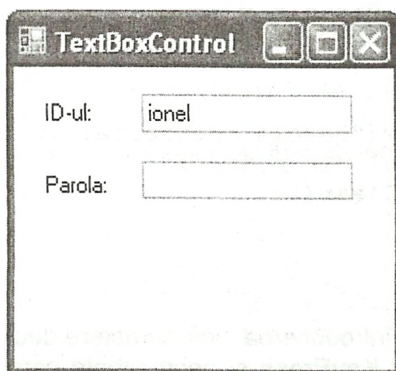
```
private void autentifButton_Click(object sender,
                                   EventArgs e)
{
    // Dacă lungimea CNP este incorectă
    if (cnpTextBox.Text.Length != 13)
    {
         MessageBox.Show("CNP incorect!");
         // Se şterge textul din control
         cnpTextBox.Clear();
    }
```

```
        else
            MessageBox.Show("Autentificare cu succes!");
    }
```

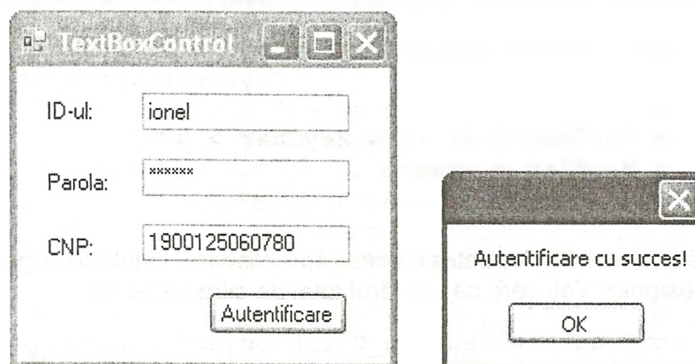
9. Compilați şi rulați cu **F5**.

Utilizatorul va completa primul câmp de text cu ID-ul "*Ionel*", după care va apăsa **Tab**, al doilea **câmp** cu parola "*parola*", după care apasă **Enter**, iar in al treilea va trece un număr format din **13** cifre, apoi va face click pe buton. Aplicația verifică corectitudinea ID-ului, a parolei şi a CNP-ului, într-un mod, desigur rudimentar, dar suficient pentru ceea ce ne-am propus.

La rulare, înainte de introducerea parolei, avem:



După introducerea parolei "*parola*" şi a unui ID format din 13 cifre:



Veți constată că acele caractere tastate care nu sunt cifre nu apar în *TextBox*-ul pentru *CNP*.

De reținut:

- Proprietatea **PasswordChar** stabileşte caracterul care maschează textul introdus de utilizator.

- Proprietatăţile **Enabled** şi **Visible** au valorile `true` sau `false`. Implicit este `true`. Valoarea `false` pentru **Enabled**, aduce controlul în stare inactivă, în timp ce `false` pentru **Visible** face controlul invizibil pe formă. Sunt caracteristici ale tuturor controalelor, întrucât se moştenenesc de la clasa de bază `Control`.
- Evenimentele **PreviewKeyDown** sau **KeyPress** au fost alese şi în funcţie de abilităţile parametrilor *handler*-elor lor. De exemplu, obiectul `KeyPressEventArgs`, parametru al metodei `cnpTextBox_KeyPress` conţine proprietatea `KeyChar`, necesară identificării codului *ASCII* al caracterului introdus.
- Metoda `Focus()`, moştenită de la clasa `Control`, setează focusul pe controlul curent.

Elemente de Stil

Este recomandabil ca în aplicaţiile în care folosiţi mai multe controale, să introduceţi propriile nume pentru referinţe şi tipuri. Numele trebuie să fie sugestive, adică să indice menirea şi tipul controlului. Exemplu: ***`idTextBox`****,* ***`parolaTextBox`****, etc. Numele claselor trebuie să înceapă cu literă mare, de exemplu* ***`TextBoxExample`****. În felul acesta veţi distinge uşor referinţele şi tipurile.*

Focusul Controalelor

Controlul care primeşte intrările de la tastatură este cel care "*are **focusul***". În oricare moment, un singur control al aplicaţiei poate avea focusul. Schimbarea focusului se face acţionând tastele săgeţi, tasta **Tab**, sau mouse-ul.

Ordinea în care controalele primesc focusul se stabileşte programatic cu ajutorul proprietăţii **TabIndex**. Controlul cu valoarea **0** pentru **TabIndex** primeşte primul focusul. Visual C# 2008 permite stabilirea *design time* a focusului controlalor. În meniul **View**, alegeţi **Tab Order**, apoi faceţi click pe etichete pentru modificări.

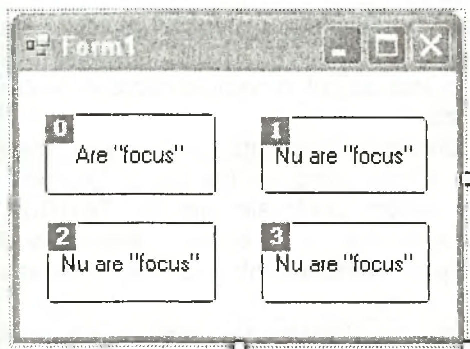


Figura 7.1 ***Focus stabilit design time***

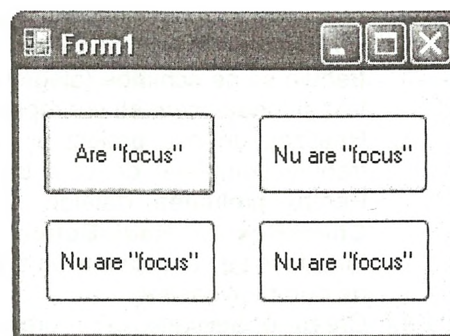


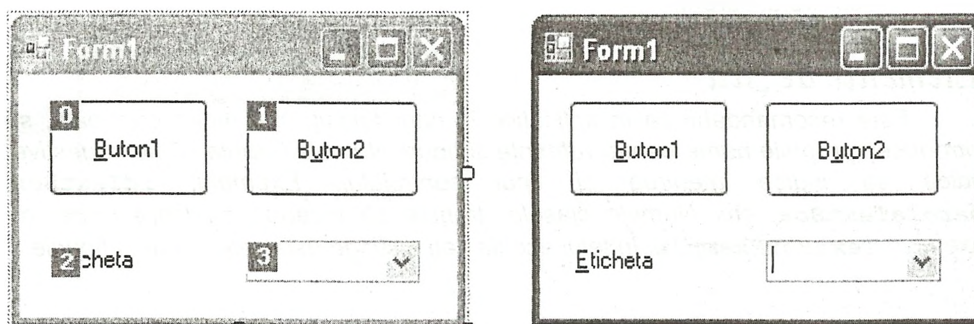
Figura 7.2 ***Focusul la rulare***

Chei de accesare a controalelor

În timpul rulării, un control poate primi în mod direct focusul de la tastatură, sărind peste ordinea stabilită prin **TabIndex**, cu ajutorul unei combinații de taste: *Alt + Caracter*. Este nevoie pentru aceasta ca valoarea proprietatății **Text** să conțină caracterul ampersand. Pentru exemplul de mai jos, proprietatatea **Text** are valorile: `&Buton1`, `B&uton2`, `&Eticheta`.

Label-urile au **TabIndex**, chiar dacă ele nu pot primi focusul. Dacă le accesați cu o combinație de chei, controlul care urmează etichetei în ordinea de indexare, va primi focusul.

După combinația *Alt+E*, **ComboBox**-ul va primi focusul:



Controalele invizibile sau cele dezactivate nu pot fi accesate cu tasta **Tab**, şi nu pot primi focusul, chiar dacă li s-a asociat un index de tabulare *design time* sau *run time*.

Probleme propuse

1. Realizați o aplicație care foloseşte un text box ca editor de text. Explorați proprietățile clasei **TextBox** şi testați-le pentru acest control (Exemplu: culoarea fontului, dimensiune, culoare *backgound*, etc).
2. Realizați o aplicație care utilizează un **TextBox** pentru introducerea textului. La fiecare caracter introdus, culoarea fundalului şi a textului trebuie să se schimbe (alegeți câte o listă de culori complementare pentru text şi fundal, care se vor repeta ciclic).
3. Realizați un mic proiect a cărui fereastră reprezintă un formular care trebuie completat de căte un elev, atunci când se înscrie la facultate. Pentru preluarea datelor, se vor utiliza controale de tip **TextBox**, **CheckBox** şi **RadioButton**. La apăsarea unui buton, toate datele intruduse se vor lista într-un control de tip **TextBox**, într-o formatare aleasă de dumneavoastră.
4. Creați o aplicație care preia adresa si numărul de telefon ale unei persoane (introduse în controale de tip text diferite) şi afişează într-un *MessageBox* toate informațiile culese.
5. Implementați o aplicație care preia o comandă de produse efectuată de către un client şi afişează toate elemente sale

Controalele MenuStrip şi ContextMenuStrip

Meniurile sunt controale familiare în peisajul aplicaţiilor de zi cu zi. De exemplu, utilitarul clasic ***Notepad*** are un meniu din care utilizatorul poate seta diverse acţiuni de formatare a textului sau de editare. Cu ajutorul controlului de tip container **MenuStrip** se pot crea deosebit de uşor asemenea meniuri.

Meniurile de context definite cu **ContextMenuStrip** se accesează cu click drept şi sunt folosite deseori în editare de text sau oriunde doriţi să aveţi acces rapid la anumite opţiuni, cum ar fi *Cut*, *Copy*, *Paste*. Un meniu contextual se asignează întotdeauna unui control, acelui control pe care faceţi click drept.

Principalii membri ai clasei MenuStrip

Clasa este bogată în proprietăţi, metode şi evenimente. Din fericire, nu aveţi nevoie decât extrem de rar să modificaţi valorile implicite ale proprietăţilor. De asemenea, din mulţimea de evenimente, trataţi de regulă doar evenimentul Click.

Urmează o prezentare selectivă a membrilor:

Proprietăţi :

`Anchor`	-	Determină modul în care se redimensionează controlul în raport cu containerul său.
`BackgroungImage`	-	Returnează sau setează imaginea afişată în control
`ClientSize`	-	Returnează sau setează înălţimea şi lăţimea zonei client a controlului.
`ContextMenu`	-	Setează meniul de context asociat
`Text`	-	Reprezintă textul asociat controlului

Metode:

`FindForm()`	-	Returnează o referinţă la forma pe care se găseşte controlul
`Hide()`	-	Ascunde controlul
`Show()`	-	Afişează controlul

Evenimente:

`Click`	-	Se declanşează la click în control
`MenuActivate`	-	Se declanşează cînd utilizatorul accesează meniul cu tastatura sau mouse-ul
`Leave`	-	Se declanşează cînd focusul părăseşte controlul
`TextChanged`	-	Se declanşează când proprietatea `Text` se schimbă.

Aplicația *MenuExample*

Proiectul pe care îl vom realiza pentru acomodarea cu cele două controale implementează:

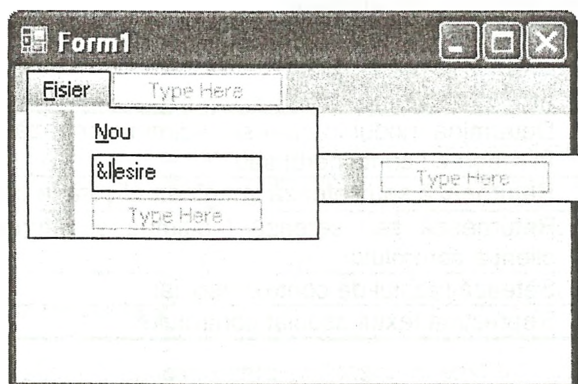
- Un editor de text rudimentar cu ajutorul unui control **TextBox**.
- Operațiile *cut/copy/paste* . Aceste sunt accesibile atât dintr-un meniu cât şi dintr-un meniu contextual.

Urmați paşii:

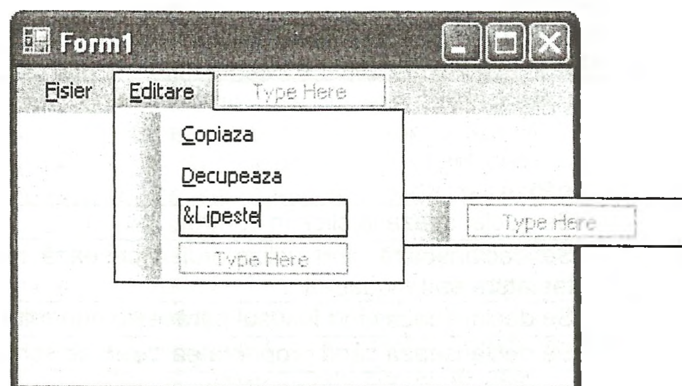
1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *MenuExample*.

2. Din **Toolbox** trageţi cu ajutorul mouse-ului un control **MenuStrip** pe suprafața formei.

3. Faceți click pe caseta controlului, acolo unde scrie *Type Here*, şi creați un prim meniu numit **Fisier**, cu opțiunile ***Nou*** şi ***Iesire***:

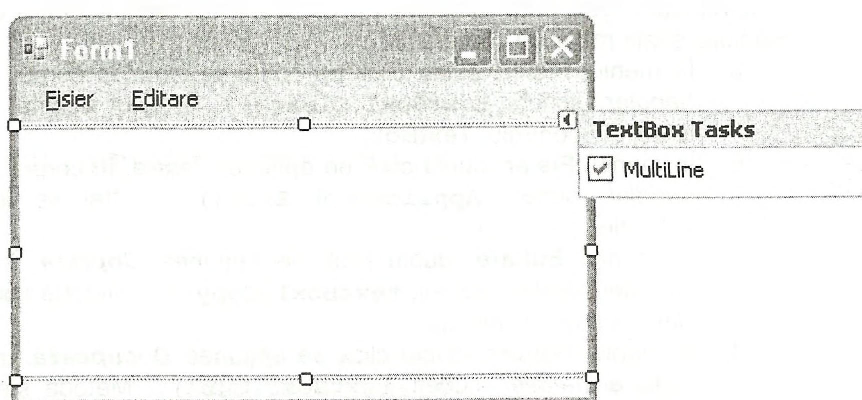


4. Creați un al doilea meniu cu numele **Editare** şi opțiunile ***Copiaza***, ***Decupeaza*** şi ***Lipeste***:

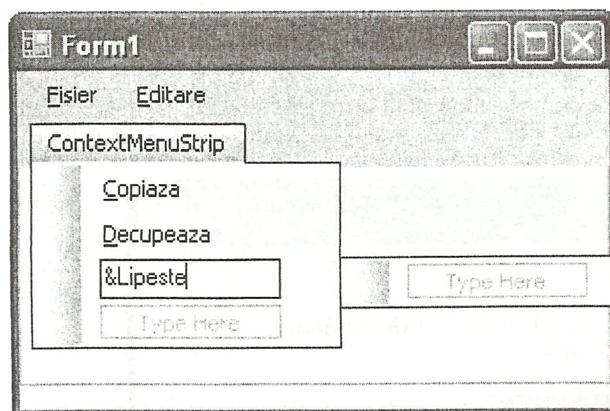


5. Din **Toolbox** alegeți un contrlol **TextBox** şi plasați-l pe suprafața formei.

6. Setați pentru **TextBox** proprietatea **Multiline** la `true`. Acest lucru se poate face din **Properties** sau mai simplu cu ajutorul meniului contextual care apare prin click pe săgeata din colțul dreapta sus a controlului:



7. Din **Toolbox** trageți pe suprafața ferestrei aplicației un control de tip **ContextMenuStrip**. Selectați controlul şi introduceți opțiunile: **Copiaza**, **Decupeaza**, **Lipeste**:

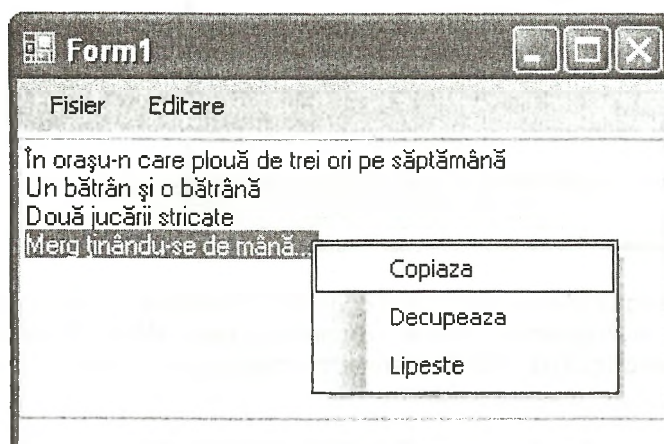


8. Acum veți asigna meniul de context controlului **TextBox**. Selectați controlul **TextBox** şi în **Properties** atribuiți proprietății **ContextMenuStrip** valoarea `contextMenuStrip1`, adică referința la obiectul de tip meniu contextual al aplicației:



9. În continuare, veți trata evenimentul **Click** pentru fiecare dintre opțiunile meniului şi ale meniului contextual:
   a. În meniul **Fisier**, dublu click pe opțiunea ***Nou***. În corpul metodei *handler*, scrieți: `textBox1.Clear();`. În felul acesta, ştergeți textul din controlul **TextBox**.
   b. În meniul **Fisier**, dublu click pe opțiunea ***Iesire***. În corpul metodei *handler*, scrieți: `Application.Exit();`. Astfel se iese din aplicație.
   c. În meniul **Editare**, dublu click pe opțiunea ***Copiaza***. În corpul metodei *handler*, scrieți: `textBox1.Copy();`. Metoda copiază în *clipboard* textul selectat.
   d. În meniul **Editare**, dublu click pe opțiunea ***Decupează***. În corpul metodei *handler*, scrieți: `textBox1.Cut();`. Metoda şterge din control textul selectat şi îl copiază în *clipboard*.
   e. În meniul **Editare**, dublu click pe opțiunea ***Lipeste***. În corpul metodei *handler*, scrieți: `textBox1.Paste();`. Metoda scrie în control textul memorat în *clipboard*.
   f. Selectați meniul contextual din bara gri, din partea de jos a *Form Designer*-ului. Veți face dublu click pe fiecare opțiune a meniului contextual şi veți trata evenimentul **Click** în aceeaşi manieră în care ați procedat cu opțiunile ***Copiază***, ***Decupeaza*** şi ***Lipeste*** ale meniului aplicației.

10. Compilați şi rulați cu **F5**.



Forme

O formă este un tip special de control care reprezintă o fereastră. Este folosită ca suport pentru alte controale. Prin setarea proprietăţilor se obţin diferite tipuri de ferestre, având diverse dimensiuni, aparenţe, stiluri şi culori. Formele sunt instanţe ale clasei **`Form`**.

Principalii membri ai clasei Form

Din galeria de membrii ai acestei clase, prezentăm:

Proprietăţi :

`ClientSize`	- Returnează şi setează suprafaţa utilă a formei, pe care se pot plasa controale sau se poate desena.
`DialogResult`	- Setează valoarea pe care o returnează o formă când funcţionează ca dialog modal.
`Modal`	- Stabileşte dacă o formă se afişează sau nu ca dialog modal.
`Size`	- Returnează sau modifică dimensiunie formei.

Metode:

`Activate()`	- Activează forma şi îi transferă *focusul*
`Close()`	- Închide forma
`Invalidate()`	- Invalidează întreaga suprafaţă a formei, fapt care cauzează redesenarea ei
`Show()`	- Afişează forma
`ShowDialog()`	- Afişează forma ca dialog modal
`Text`	- Specifică un text în bara de titlu

Evenimente:

`Closing`	- Se declanşează când forma se închide
`Load`	- Se declanşează înainte ca forma să fie afişată pentru prima oară.

Crearea formelor

1. Crearea programatică a formelor

În mod programatic, o formă se crează foarte simplu:

```
Form f = new Form();      // Se instanţiază forma
f.Show();                 // Se afişează
```

O metodă obişnuită de creare a unei forme este de a defini o clasă derivată din clasa **Form**:

```
class FormaMea : Form
{
    public FormaMea() // Constructorul clasei
    {
    }
    // Alţi membri ai clasei (câmpuri, proprietăţi, metode)
}
```

Dintr-o altă metodă a aplicaţei, instanţiaţi forma:

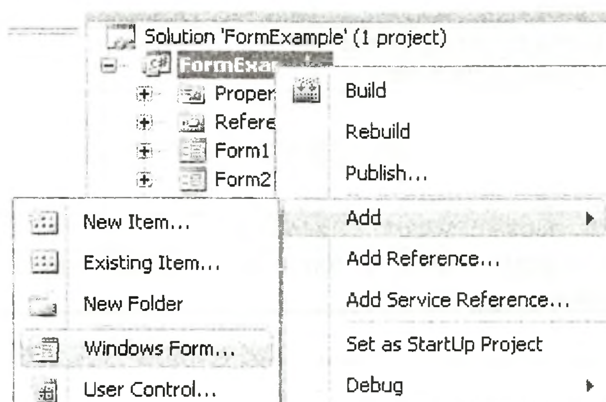
```
FormaMea f = new FormaMea();
f.Show();
```

2. Crearea formelor cu ajutorul *Form Designer*-ului

La crearea unui nou proiect de tip *Windows Forms*, mediul integrat generează o clasă cu numele implicit **Form1**, iar această clasă se instanţiază în **Main()**:

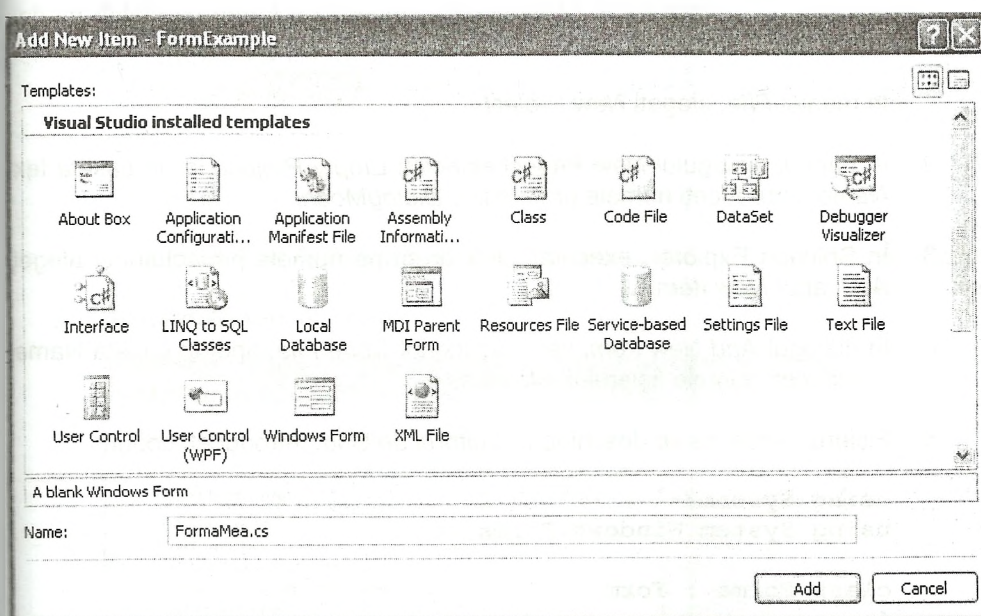
```
Application.Run(new Form1());
```

Puteţi adăuga proiectului oricâte alte forme, în felul următor:
În ***Solution Explorer***, acţionaţi click drept pe numele proiectului, alegeţi *Add*, apoi *Windows Form...* :



Ca variantă alternativă, click drept pe numele proiectului, alegeţi *Add*, apoi *New Item....*

În fereastra *Add New Item – [nume proiect]* selectaţi iconul *Windows Form* şi introduceţi un nume pentru noua formă:



Dialoguri modale şi dialoguri nemodale

Una dintre forme este fereastra de bază a aplicaţiei. Celelalte forme reprezintă ferestre care pot fi făcute să apară la un moment dat.

Dialoguri modale

Anumite tipuri de ferestre se prezintă ca dialoguri cu utilizatorul. Acesta introduce date în controale, pe care apoi le validează sau nu (apăsând de exemplu butoanele **OK** sau **Cancel**). Acest tip de fereastră dialog nu permite utilizatorului să acceseze alte ferestre ale aplicaţiei până când dialogul nu este închis. De exemplu, în *Microsoft Word*, când alegeţi ***Open*** în meniul ***File***, se deschide un dialog care vă cere să alegeţi un fişier. Fereastra blochează aplicaţia şi nu puteţi întreprinde nici o altă acţiune în editor, până în momentul în care aţi închis dialogul. Un asemenea tip de dialog, se numeşte **dialog modal**.
Un dialog modal este o formă care se deschide cu metoda `ShowDialog()`:

```
class Forma : Form
{
    // membrii clasei
}

Forma f = new Forma();
f.ShowDialog();
```

Aplicația *DialogModal*

Realizăm un proiect de tip consolă care lansează o formă ca dialog modal. Urmați pașii:

1. În meniul ***File***, alegeți *New Project*.

2. În panoul dialogului *New Project* selectați *Empty Project*, iar în caseta text *Name*, introduceți numele proiectului: *DialogModal*.

3. În Solution Explorer, executați click drept pe numele proiectului și alegeți *Add*, apoi *New Item...*.

4. În dialogul *Add New Item*, selectați iconul *Code File*, apoi în caseta Name, introduceți numele fișierului: *Modal.cs*.

5. Fișierul *Modal.cs* se deschide în *Editorul de Cod*. Introduceți codul:

```
using System;
using System.Windows.Forms;

class Forma : Form
{
}

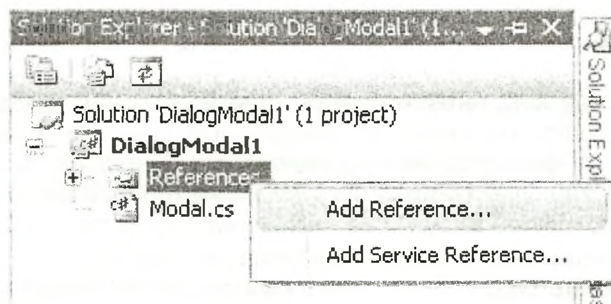
class Pogram
{
    public static void Main()
    {
        Forma f = new Forma();    // Instanţiere

         // Text în bara de titlu
        f.Text = "Dialog Modal";

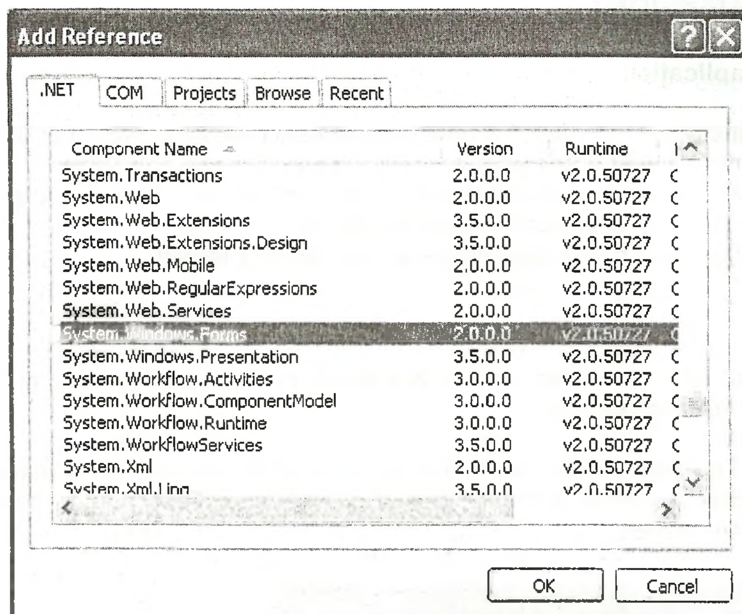
        // Dialogul modal devine vizibil
        f.ShowDialog();

        Application.Run();        // Lansează aplicaţia
    }
}
```

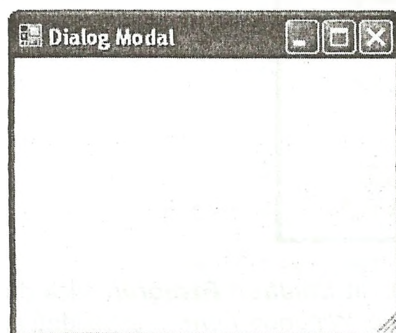
6. Încă nu putem compila, pentru că proiectul de tip consolă nu introduce în mod automat referințe la bibliotecile dinamice **.NET** care implementează spațiile de nume **System.Windows** și **System.Windows.Forms**. Vom aduce aceste referințe astfel: în ***Solution Explorer***, click drept pe folderul *References*:



7. În dialogul ***Add Reference***, selectaţi pe tab-ul **.NET** intrarea **System**, apoi repetaţi pasul **6** şi adăugaţi o referinţă la bibliotecile **System.Windows.Forms**:



8. Compilaţi şi rulaţi aplicaţia cu **F5**. La rulare obţineţi:



Dialoguri nemodale

Dialogurile nemodale sunt ferestre care nu întrerup execuţia aplicaţiei. Un dialog nemodal nu împiedică utilizatorul să execute acţiuni în alte ferestre ale aplicaţiei. Se utilizează în special ca bare de instrumente. Gândiţi-vă la *toolbox*-urile din *Paint* sau *Word*, sau chiar **Toolbox** din *Visual C# Express 2008*.

Afişarea se face cu metoda `Show();` La închiderea unei forme cu `Close()`, toţi membrii clasei (de exemplu controalele de pe formă) se distrug.

Ca exerciţiu de creare programatică a unui dialog nemodal, reluaţi toţi paşii aplicaţiei *DialogModal* (redenumiţi ca *DialogNemodal*) şi faceţi o singură modificare în cod: înlocuiţi `f.ShowDialog()` cu `f.Show();`

Aplicaţia *ModalDialogExample*

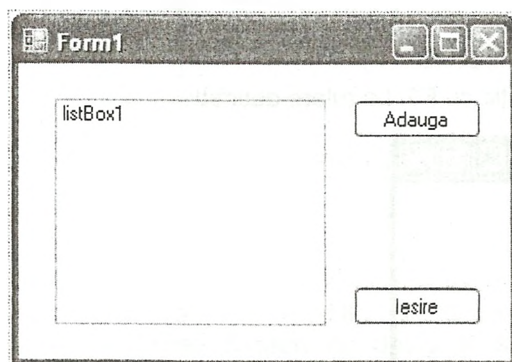
Descrierea aplicaţiei:

- Construieşte un dialog modal care se deschide la apăsarea unui buton al formei părinte a aplicaţiei.
- Datele pe care utilizatorul le introduce într-un control al dialogului modal se transferă unui control al clasei părinte.
- Anticipează modul de utilizare a unui control **ListBox**.

Urmaţi paşii:

1. Creaţi un nou proiect de tip ***Windows Forms Application***, cu numele *ModalDialogexample*.

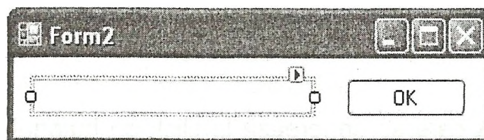
2. Din **Toolbox** plasaţi pe formă cu ajutorul mouse-ului un control de tip **ListBox** şi două butoane. Setaţi proprietatea **Text** pentru celor două butoane astfel:



3. Adăugaţi o nouă formă proiectului. În ***Solution Explorer***, click dreapta pe numele proiectului, alegeţi *Add*, apoi *Windows Form...* şi validaţi cu **OK**.

4. În ***Solution Explorer***, dublu click pe numele *Form2* (în cazul în care nu ați redenumit noua formă).

5. Aduceți din **Toolbox** un control **TextBox** şi un buton, ca în figură:



6. În ***Solution Explorer***, acționați click drept pe numele *Form2*, apoi alegeți *View Code*. În clasa **Form2** definiți membrii:

```
// Câmp care reţine textul introdus în textBox1
private string item;

// Proprietate care returnează valoarea textului
public string Item
{
    get { return item; }
}
```

7. Tratăm evenimentul **Click** generat la apăsarea butonului **OK**. Acționați dublu click pe buton. Completați codul evidențiat în **Bold**:

```
private void button1_Click(object sender, EventArgs e)
{
     // Salvăm textul introdus în control.
     item = textBox1.Text.Trim();

     // Închidem forma Form2.
     Close();
}
```

8. Dorim ca **Form2** să se închidă şi la apăsarea tastei **Enter**. Tratăm evenimentul **KeyDown** pentru controlul **TextBox**. Selectați controlul. În fereastra **Properties** apăsați butonul ***Events*** (fulgerul) şi faceți dublu click pe evenimentul **KeyDown**. Introduceți codul:

```
private void textBox1_KeyDown(object sender,
                              KeyEventArgs e)
{
    // Dacă tasta apăsată are codul tastei Enter
    if (e.KeyCode == Keys.Enter)
    {
        item = textBox1.Text.Trim(); // Memorăm textul
        Close();                     // Închidem forma
    }
}
```

9. În scopul de a ni se permite să accesăm proprietatea `Item` a clasei `Form2` din metodele clasei `Form1`, modelăm o *relaţie de conţinere* (*containment*). Definim în `Form1` un câmp privat, o referinţă la `Form2`. În ***Solution Explorer***, click drept pe *Form1* şi alegeţi *View Code*. În fişierul *Form1.cs*, în clasa `Form1`, scrieţi:

```
private Form2 f;
```

10. Tratăm evenimentul **Click** care se generează la click pe butonul *Adauga*, aflat pe forma `Form1`. În ***Solution Explorer***, faceţi dublu click *Form2*, apoi în *Form Designer* acţionaţi dublu click pe butonul *Adauga*. În corpul *handler*-ului introduceţi codul:

```
private void button1_Click(object sender, EventArgs e)
{
    f = new Form2();        // Creăm forma

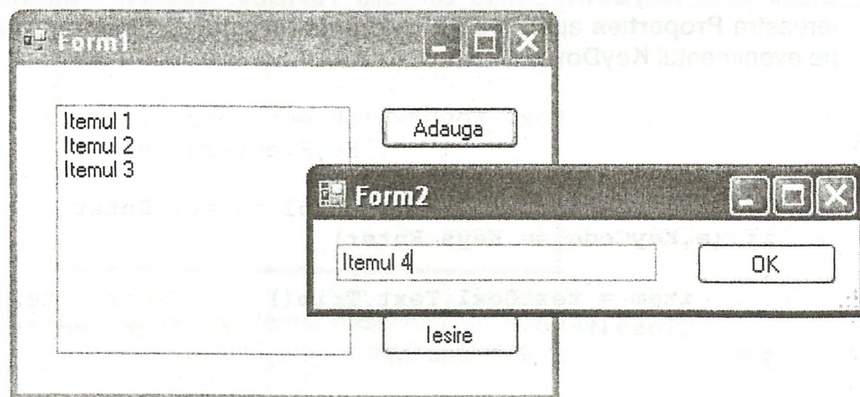
    f.ShowDialog();         // Afişăm dialogul modal
    if ( f.Item != "")      // Dacă există text
    {
        // Adăugăm itemul în listBox1
        listBox1.Items.Add(f.Item);
    }
}
```

11. Pentru ieşire din aplicaţie, faceţi dublu click pe butonul *Iesire*. În corpul *handler*-ului evenimentului Click, scrieţi:

```
Application.Exit();        // Ieşire din aplicaţie
```

12. Compilaţi şi lansaţi în execuţie cu **F5**.

La rulare, prin apăsarea butonului *Adauga*, se deschide dialogul modal ***Form2***. Când se apasă **OK** sau **Enter**, se închide dialogul *Form2*, iar textul introdus apare ca nou item în controlul **ListBox** de pe *Form1*.



Notă:

- Metoda `Trim()` a clasei `string` înlătură caracterele albe de la începutul şi sfârşitul stringului.
- Metoda `Add()` din apelul `listBox1.Items.Add(f.Item);` adaugă un item într-un controlul referit de `listBox1`.
- Metoda `Close()` închide forma.

IMPORTANT !

Când apelaţi `Close()` pentru un dialog modal, fereastra nu se distruge ci devine invizibilă. Aceasta, deoarece după închiderea ferestrei, aveţi de regulă nevoie de membrii clasei, ca în aplicaţia de faţă:

```
f.ShowDialog();        // Afişăm dialogul modal
// În acest punct fereastra este închisă!
// f.Item există şi după închiderea dialogului, deoarece
// dialogul (obiectul de tip Form2) nu este distrus,
// ci numai ascuns.
if ( f.Item != "")
{
    listBox1.Items.Add(f.Item);
}
```

De reţinut:

- Dialogurile modale se afişează cu metoda `ShowDialog()`, iar cele nemodale cu metoda `Show()`.
- Afişarea unei forme cu `Show()` se poate face alternativ prin setarea proprietăţii `Visible` la valoare `true`. Ascunderea formei se poate face şi prin setarea proprietăţii `Hide` la false.
- La închiderea formei cu `Close()`, dialogurile nemodale se distrug (se distruge instanţa clasei).
- Când o formă este afişată ca dialog modal, apelul metodei `Close()` sau un click pe butonul **Close** (butonul cu un **x** în colţul dreapta sus a formei) nu distruge forma ci o ascunde, ea putând fi făcută vizibilă ulterior.
- Codul care urmează apelului `ShowDialog()` nu se execută până când forma se închide. După închiderea dialogului modal, puteţi utiliza în continuare referinţa la clasă şi toţi membrii clasei.

Butoane de validare a datelor

Când închide un dialog modal, utilizatorul ar trebui să aibă cel puţin două opţiuni: să păstreze datele pe care le-a introdus în controale sau să renunţe la ele. Pentru aceasta, are nevoie de două butoane, să le spunem **OK** şi **Cancel**. Evident,

nu este suficient să etichetaţi cele două butoane astfel ca să lucreze în mod adecvat. Ca să joace rolurile dorite, se setează proprietatea **DialogResult** a clasei **Button**, astfel:

```
Button b1 = new Button();
Button b2 = new Button();
b1.Text = "OK";        // Poate fi şi altă etichetă
b2.Text = "Cancel";    // Poate fi şi altă etichetă

// Setăm b1 să returneze DialogResult.OK la click
b1.DialogResult = DialogResult.OK;

// Setăm b2 să returneze DialogResult.Cancel la click
b2.Dialogresult = DialogResult.Cancel;
```

În felul acesta, **b1** va fi butonul ***OK*** al formei, iar **b2** cel de închidere cu ***Cancel***. **DialogResult** este o enumerare a cărei membrii sunt: **OK**, **Cancel**, **Abort**, **Retry**, **None**, **Ignore**, **Yes**, **No**.

Cum detectăm în cod faptul că o formă a fost închisă cu ***OK*** sau cu ***Cancel*** ? Vom analiza valoarea de retur a metodei **ShowDialog()**, care este tocmai una dintre valorile membrilor enumerării şi anume, valoarea returnată de butonul apăsat. Secvenţa standard este:

```
DialogResult rez = f.ShowDialog();

switch (rez)
{
   case DialogResult.OK :
      // Ceea ce doriţi să se execute dacă s-a ieşit cu OK
      break;
   case DialogResult.Cancel :
      // Cod care se execută în caz că s-a acţionat Cancel
      break;
}
```

De ce vrem să ştim dacă forma a fost închisă cu ***OK*** sau cu ***Cancel*** ? Foarte simplu: pentru că dacă închideţi cu ***OK***, forma este doar ascunsă, şi o puteţi utiliza în continuare, ca şi datele introduse, în vreme ce dacă aţi închis cu ***Cancel***, forma se distruge împreună cu toate controalele şi datele reţinute în membrii clasei.

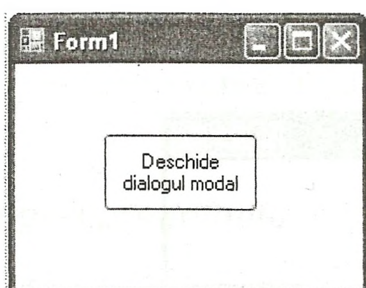
Pentru edificare, vom realiza un mic proiect de tip *Windows Forms*.

Aplicaţia *OkCancel*

Aplicaţia are două butoane, cu etichetele *OK* şi *Cancel*. Primului buton i se atribuie rolul de validare a datelor, iar celuilalt de renunţare. Amândouă închid forma. Se afişează câte un *message box* la închiderea formei pentru fiecare caz.

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *OkCancel*.

2. Din **Toolbox** plasați pe formă un buton. Atribuiți proprietății **Text** valoarea "Deschide dialogul modal".



3. Adăugați o formă nouă proiectului: în ***Solution Explorer***, click dreapta pe numele proiectului, alegeți *Add*, apoi *Windows Form*. Clasa formei se va numi *Form2*.

4. În ***Solution Explorer***, faceți dublu click pe *Form2*. Plasați pe suprafața acestei forme două butoane, pentru care setați textele *OK* şi *Cancel*.

5. Tratăm evenimentul **Load** pentru *Form2*, pentru a putea seta proprietățile **DialogResult** pentru cele două butoane. Desigur că puteți face acest lucru şi din fereastra ***Properties***. Executați dublu click pe suprafața formei. Scrieți codul:

```
private void Form2_Load(object sender, EventArgs e)
{
    // Setează button1 să returneze OK la click
    button1.DialogResult = DialogResult.OK;

    // Setează button2 să returneze Cancel la click
    button2.DialogResult = DialogResult.Cancel;
}
```

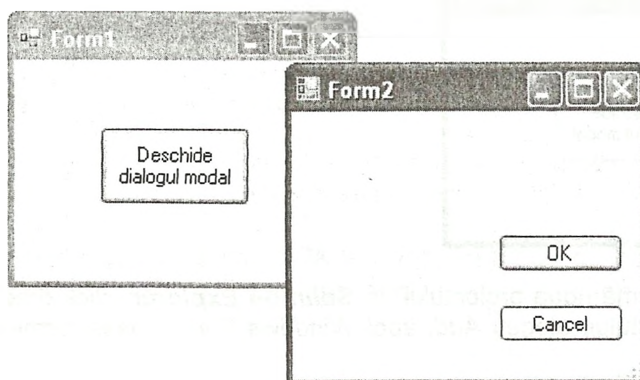
6. Tratăm evenimentul **Click** pentru butonul formei *Form1*. Faceți dublu click pe buton. În metoda *handler*, introduceți codul evidențiat în **Bold**:

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 f = new Form2();
    DialogResult rez = f.ShowDialog();

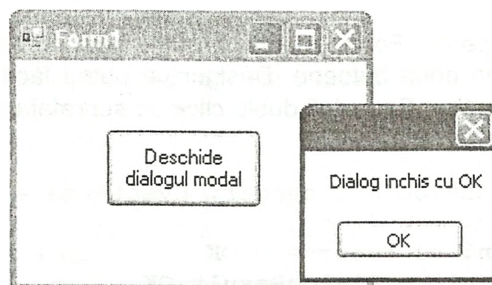
    switch (rez)
    {
       case DialogResult.OK:
           MessageBox.Show("Dialog inchis cu OK");
           break;
```

```
            case DialogResult.Cancel:
                MessageBox.Show("Dialog inchis cu Cancel");
                break;
        }
    }
```

La execuţie, înainte de închiderea *Form2*, avem:



După închiderea formei cu **OK**:



De reţinut:

- Puteţi desemna butoane care returnează valori egale cu cele ale unor membri ai enumerării **DialogResult**.
- Valoarea returnată la acţionarea click asupra unui asemenea buton este aceeaşi pe care o returnează metoda `ShowDialog()`. Acest lucru permite ca ulterior închiderii formei să se dirijeze fluxul de execuţie al programului în funcţie de rezultatul întors de formă.

Probleme propuse

1. Testaţi metoda **MessageBox.Show**(*text, titlu, butoane*) cu toate butoanele diponibile.

2. Testaţi metoda `MessageBox.Show()` în versiunea cu **4** parametri, folosind pe rând toate butoanele şi toate iconurile disponibile.

3. Realizaţi o aplicaţie care deschide un dialog modal la acţionarea unui buton de pe forma principală. Dialogul modal conţine un formular realizat în principal cu controale *text box*, în care se reţin date personale despre clienţii unei firme. La închiderea dialogului modal cu **OK**, datele se afişează într-un format ales de dumneavoastră, într-un control *text box* de pe forma părinte. Dacă dialogul modal s-a închis cu butonul **Cancel**, atunci nu se salvează datele.

Dialoguri predefinite

.NET oferă câteva dialoguri predefinite pe care le puteţi invoca în aplicaţii. Dialogurile sunt forme. Când în *Notepad* sau *Microsoft Word* acţionaţi **File -> Open**, **File -> Save As** sau **File -> Page Setup**, invocaţi de fapt unul dintre dialogurile standard *Windows*.

Clasele **.NET** care reprezintă dialogurile predefinite sunt: **MessageBox**, **OpenFileDialog**, **SaveFileDialog**, **FontDialog**, **ColorDialog**, **PageSetupDialog**, **PrintDialog** şi **PrintPreviewDialog**.

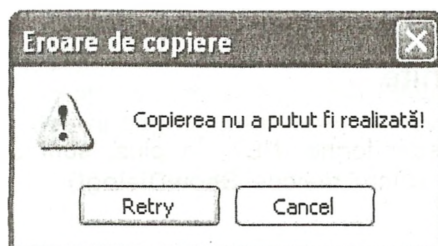
Toate aceste dialoguri sunt **dialoguri modale**.

Descrierea dialogurilor predefinite

MessageBox este cel mai comun dialog predefinit. Clasa defineşte metoda statică `Show()` în mai multe versiuni. Cea mai simplă versiune, cu un singur parametru, a fost deja folosită în această lucrare. În exemplu se apelează o metodă `Show()` cu patru parametri:

```
MessageBox.Show("Copierea nu a putut fi realizată!",
                 "Eroare de copiere",
                 MessageBoxButtons.RetryCancel,
                 MessageBoxIcon.Exclamation);
```

Efectul pe ecran este :



Icon-urile pe care le poate afişa un *message box* sunt:

Asterisc, **Error**, **Exclamation**, **Hand**, **Information**, **None**, **Question**, **Stop** şi **Warning**. Aceştia sunt membrii ai enumerării `MessageBoxIcon`, iar semnificaiția lor este evidentă.

Butoanele pe care le poate afişa un *message box* sunt: **AbortRetryIgnore** (3 butoane), **OK**, **OKCancel** (2 butoane), **YesNo** (2 butoane), **YesNoCancel** (3 butoane). Aceste entități sunt membri ai enumerării `MessageBoxButtons`.
În aplicația pe care o vom realiza în cadrul temei ***MDI – Multiple Document Interface*** vom arăta cum dați funcționalitate butoanelor de pe formă.

OpenFileDialog moşteneşte clasa **FileDialog**. Dialogul (forma) este o instanță a clasei. Permite utilizatorului să selecteze un fişier pentru a fi deschis. Atenție, dialogul nu deschide fişierul! Acest lucru cere un mic efort de programare, aşa cum vom vedea în subcapitolul următor. Numele fişierului selectat se atribuie în mod automat proprietății **FileName**. Formatul fişierelor care trebuie deschise se stabilesc cu ajutorul proprietății **Filter**.

SaveAsDialog moşteneşte clasa **FileDialog**. Dialogul este folosit în scopul alegerii unei locații pentru salvarea unui fişier. Dialogul nu salvează fişierul. Numele fişierului selectat se atribuie în mod automat proprietății **FileName**. Clasa poate crea un nou fişier sau poate suprascrie unul existent.

FontDialog derivă din clasa **CommonDialog**. Utilizatorul poate alege un font dintre cele instalate pe calculator prin simpla selectare cu mouse-ul.

PageSetupDialog are ca şi clasă de bază directă **CommonDialog**. Utilizatorul poate stabili setări pentru pagină, cum ar fi marginile, formatul, etc.

PrintDialog derivă din clasa **CommonDialog**. Dialogul permite alegerea unei imprimante, setarea formatului de imprimare, alegerea unei porțiuni de document care va fi printată şi invocarea imprimantei.

PrintPreviewDialog este derivată din clasa **Form**. Dialogul permite examinarea unui document înainte de imprimare.

Indicație:
Alegeți un utilitar, cum este *Microsoft Word* şi revedeți toate aceste dialoguri, accesându-le pe rând din meniul **File**. Merită să o faceți, pentru că acum le vedeți cu ochii programatorului.

Afişarea dialogurilor predefinite

Dialogurile standard Windows sunt forme .NET. În plus, sunt dialoguri modale. Aşadar, afişarea lor se face cu ajutorul metodei **ShowDialog()**:

Exemplu:

Iată cum afişați dialogul **OpenFileDialog**:

```
if (openFileDialog.ShowDialog() ==
                    System.Windows.Forms.DialogResult.OK)
{
    // Cod care deschide fisierul
}
```

`openFileDialog` este o referință la un obiect de tip `OpenFileDialog`. Referința se crează în mod automat în momentul în care din **Toolbox** alegeți un control **OpenFileDialog** şi îl plasați pe suprafața formei.

Valoarea de retur a metodei este egală cu unul dintre membrii enumerării `DialogResult`. Aceştia sunt: **OK**, **Cancel**, **Abort**, **Retry**, **None**, **Ignore**, **Yes**, **No**. Fiecăreia dintre aceste valori îi corespunde un buton pe suprafața dialogului. În mod implicit, dialogurile predefinite afişează butoanele **OK** şi **Cancel**, însă prin setarea proprietăților corespunzătoare ale dialogului, le puteți afişa şi pe celelalte.

Codul de mai sus se interpretează astfel: `ShowDialog()` afişează dialogul **OpenFileDialog**. Dacă utilizatorul apasă butonul **OK**, atunci se execută codul dintre acolade. Ce anume trebuie să scrieți, vom vedea la secțiunea **"Controlul RichTextBox"**.

Pentru afişarea celorlalte controale predefinite, se procedează similar, cu singura diferență că veți înlocui referința `openFileDialog` cu una dintre referințele: `saveFileDialog`, `printDialog`, `pageSetupDialog`, etc.

Observație:

Toate cele şapte dialoguri predefinite sunt ***dialoguri modale***, deoarece utilizează metoda `ShowDialog()`. Un dialog modal întrerupe aplicația, iar utilizatorul nu poate continua până în momentul în care dialogul a fost închis.

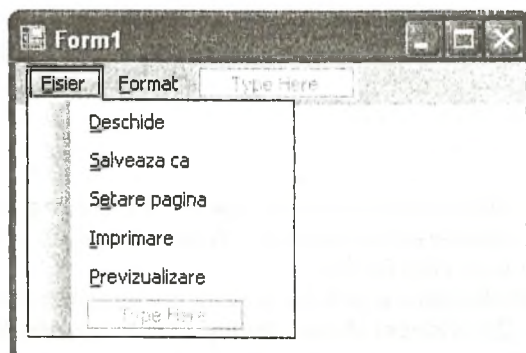
Aplicația *DialoguriPredefinite*

Proiectul pe care îl propunem este o aplicație cu un meniu din care utilizatorul poate selecta opțiuni de deschidere a dialogurilor predefinite. Pentru unele dialoguri, se va scrie şi cod suplimentar, mai ales referitor la setarea proprietăților. Vestea bună este că veți scrie întotdeauna aproape la fel atunci când lucrați cu dialoguri predefinite.

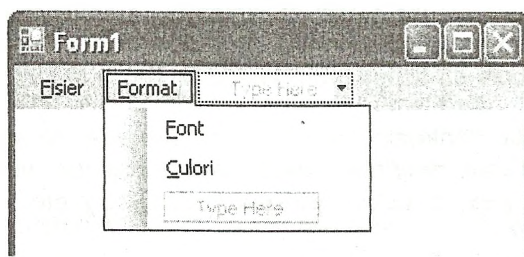
1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *DialoguriPredefinite*.

2. Din **Toolbox** plasați cu ajutorul mouse-ului un control **MenuStrip** pe suprafața formei.

3. Creați un prim meniu numit **Fisier**, cu opțiunile **Deschide**, **Salveaza ca**, **Setare pagina**, **Imprimare** şi **Previzualizare**:



4. Creați un al doilea submeniu numit **Format**, cu opțiunile **Font** şi **Culori**:



5. Din **Toolbox** plasați pe suprafața formei următoarele controale: **OpenFileDialog**, **SaveFileDialog**, **PageSetupDialog**, **PrintDialog**, **PrintPreviewDialog**, **PrintDocument**, **FontDialog**, **ColorDialog**. În "*tray*"-ul designerului de forme veți remarca imediat referințele la aceste obiecte, care au fost adăugate în cod.

6. În continuare veți trata evenimentul **Click** pentru fiecare dintre opțiunile meniului. Faceți dublu click pe opțiunea **Deschide**. În *handler*-ul evenimentului, scrieți codul:

```
private void deschideToolStripMenuItem_Click
                                (object sender, EventArgs e)
{
     // Titlul dialogului
     openFileDialog1.Title = "Deschide Fisier";

     // Setează tipurile de fişiere care apar
     // în combobox-ul "Files of Type"
     openFileDialog1.Filter =
"Fisiere Rich Text (*.rtf)|*.rtf|Fisiere Text (*.txt)|*.txt";

     // În combobox-ul File Name nu vrem să apară la
     // deschiderea dialogului nici un nume de fişier
     openFileDialog1.FileName = "";
```

```
    // Directorul care se deschide în mod implicit
    openFileDialog1.InitialDirectory = "MyDocuments";

    // Afişează o atenţionare dacă utilizatorul
    // specifică un fişier care nu există
    openFileDialog1.CheckFileExists = true;

    // Deschide dialogul OpenFileDialog
    if (openFileDialog1.ShowDialog() ==
        System.Windows.Forms.DialogResult.OK)
    {
        // Cod care deschide fişierul
    }
}
```

7. Dublu click pe opţiunea **Salvează ca**. În metoda de tratare, scrieţi:

```
private void salveazaCaToolStripMenuItem_Click
                         (object sender, EventArgs e)
{
    // Titlul dialogului
    saveFileDialog1.Title = "Salveaza Fisierul";

    // Extensia implicită de salvare a fişierelor
    saveFileDialog1.DefaultExt = ".rtf";

    // Atenţionare dacă încercaţi să suprascrieţi
    saveFileDialog1.OverwritePrompt = true;

    // Deschide dialogul SaveFileDialog
    if (saveFileDialog1.ShowDialog() ==
        System.Windows.Forms.DialogResult.OK)
    {
        // Cod care salvează fişierul
    }
}
```

8. Dublu click pe opţiunea **Setare pagina**. În metoda de tratare, scrieţi:

```
private void setarePaginaToolStripMenuItem_Click
                         (object sender, EventArgs e)
{
    // Se preiau setările din printDocument1
    pageSetupDialog1.Document = printDocument1;

    // Setările de pagină
    pageSetupDialog1.PageSettings =
                  printDocument1.DefaultPageSettings;
```

```
        // Diverse secţiuni ale dialogului devin active sau
        // inactive. De fapt, valoarea implicită a acestor
        // proprietăţi este true
        pageSetupDialog1.AllowMargins = true;
        pageSetupDialog1.AllowPaper = true;
        pageSetupDialog1.AllowOrientation = true;
        pageSetupDialog1.AllowPrinter = true;
        pageSetupDialog1.ShowNetwork = true;
        pageSetupDialog1.EnableMetric = false;
        pageSetupDialog1.ShowHelp = false;

        // Se deschide dialogul şi se preiau setările
        // de pagină în obiectul de tip PrintDocument
        if (pageSetupDialog1.ShowDialog() ==
                     System.Windows.Forms.DialogResult.OK)
           printDocument1.DefaultPageSettings =
                            pageSetupDialog1.PageSettings;
    }
```

9. Faceţi dublu click pe opţiunea **Imprimare**. În *handler* scrieţi codul:

```
    private void imprimareToolStripMenuItem_Click
                              (object sender, EventArgs e)
    {
         // Se preiau setările de printare din
         // obiectul de tip PrintDocument
         printDialog1.Document = printDocument1;

         // Se deschide dialogul PrintDialog
         if (printDialog1.ShowDialog() ==
                     System.Windows.Forms.DialogResult.OK)
         {
             // Startează procesul de printare
             printDocument1.Print();

             // Cod care transformă documentul
             // în imagine şi îl printează
             // ...
         }
    }
```

10. Dublu click pe opţiunea **Previzualizare**. În *handler*, scrieţi codul:

```
    private void previzualizareToolStripMenuItem_Click
                             (object sender, EventArgs e)
    {
         if (printPreviewDialog1.ShowDialog() ==
                   System.Windows.Forms.DialogResult.OK)
         {
             // Cod care afişează documentul în dialog
         }
    }
```

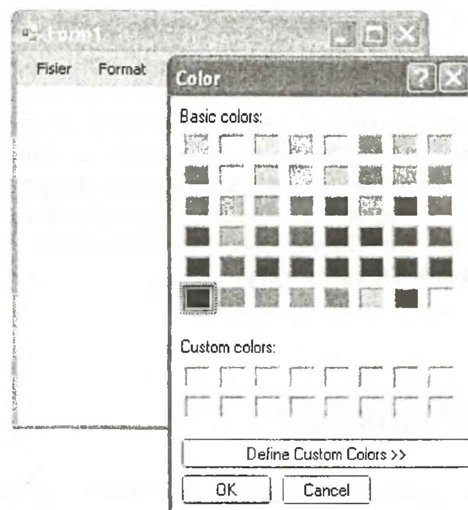
11. Dublu click pe opțiunea **Font**. În *handler*-ul evenimentului, scrieți codul:

```
private void fontToolStripMenuItem1_Click
                              (object sender, EventArgs e)
{
     // Se deschide dialogul FontDialog
     if (fontDialog1.ShowDialog() ==
             System.Windows.Forms.DialogResult.OK)
     {
         // Cod care reafişează documentul cu
         // fontul ales de utilizator
     }
}
```

12. Dublu click pe opțiunea **Culori**. În *handler*-ul evenimentului, scrieți codul:

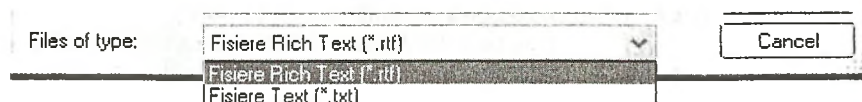
```
private void culoriToolStripMenuItem_Click
                                (object sender, EventArgs e)
{
     // Se deschide dialogul ColorDialog
     if (colorDialog1.ShowDialog() ==
               System.Windows.Forms.DialogResult.OK)
     {
         // Cod care actualizează culorile in document
     }
}
```

13. Compilați şi rulați cu **F5**.



Comentarii:

- Proprietatea `Filter` este comună dialogurilor **OpenFileDialog** şi **SaveAsDialog**, întrucât este moştenită de la clasa de bază comună, `FileDialog`. Caracterul `'|'` (pipe) delimitează tipurile de fişiere pe care le puteţi selecta, dar şi intrările în *combobox* : `"Fisiere Rich Text (*.rtf)|*.rtf|Fisiere Text (*.txt)|*.txt";`



- Dialogurile predefinite se afişează cu metoda `ShowDialog()`, iar tipul de retur al acestei metodei se analizează pentru a vedea dacă utilizatorul doreşte să păstreze setările pe care le-a făcut sau vrea să renunţe la ele.
- Reţineţi că `OpenFileDialog` nu deschide un fişier (document) ci numai vă permite să-l alegeţi. Pentru deschidere aveţi nevoie de un control suplimentar, capabil să afişeze conţinutul fişierului, aşa cum este un **TextBox** sau un **RichTextBox**. Aceleaşi consideraţii sunt valabile şi pentru celelalte dialoguri predefinite.

Notă:

În ce priveşte printarea efectivă a unui document, aceasta se face prin apelul metodei `Print()` a clasei `PrintDocument`. Clasa `PrintDocument` este definită în spaţiul de nume **System.Drawing.Printing**. Obiectul de tip `PrintDocument` preia setările de pagină stabilite de către utilizator în dialogul **PageSetupDialog**, şi trimite documentul la imprimantă. Codul necesar poate fi găsit în documentaţia clasei **PrintDocument** sau în articole din **MSDN** (*Microsoft Developer Network*).

Sfat practic

Să aveţi în permanenţă deschis Helpul mediului integrat. Acolo găsiţi imediat clasele cu care lucraţi, cu descrierea completă a câmpurilor, proprietăţilor, metodelor şi evenimentelor lor, dar şi cu exemple de cod care vă pot ajuta. Veţi descoperi foarte repede membrii ai claselor care vă pot fi utili.

*Reamintim locaţia **Bibliotecii de Clase: Help -> Contents -> .NET Framework SDK -> .NET Framework Class Library**.*

MDI – Multiple Document Interface

O aplicaţie **MDI** se lansează cu o singură fereastră container care reprezintă întreaga aplicaţie.

În interiorul ferestrei container se pot deschide multiple ferestre *child*. Gândiți-vă la *Microsoft Word*. Ferestrele interioare reprezintă diferite documente pe care utilizatorul le poate edita în acelaşi timp sau diferite vederi ale aceloraşi date.

Aplicațiile **MDI** diferă de cele de tip **SDI** (*Single document interface*) prin aceea că **SDI** au o singură fereastră copil. *Notepad* este un asemenea exemplu. Dacă doriți să editați două documente, trebuie să deschideți două instanțe *Notepad*.

Atât fereastra părinte cât şi ferestrele copil sunt de fapt forme **.NET**. O formă devine container în momentul în care proprietatea **`IsMdiContainer`** este setată **`true`**.

Aplicația *MdiExample*

Aplicația implementează:

- Deschiderea şi închiderea ferestrelor *child*.
- Minimizarea şi maximizarea ferestrelor.
- Aranjarea ferestrelor.
- Tratarea mesajului de atenționare la ieşirea din aplicație.

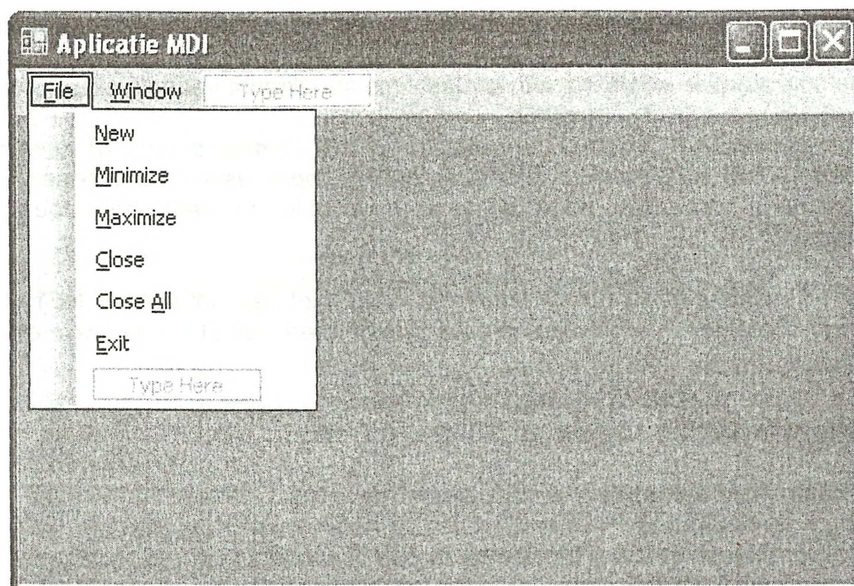
Urmați paşii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *MdiExample*.

2. Setați forma ca şi container **MDI**. Pentru aceasta, în fereastra ***Properties*** atribuiți proprietății **`IsMdiContainer`** valoare **`true`**.

3. Din **Toolbox** plasați cu ajutorul mouse-ului un control **MenuStrip** pe suprafața formei.

4. Creați un prim meniu numit **File**, cu opțiunile **New**, **Minimize**, **Maximize**, **Close**, **Close All** şi **Exit**. Creați un al doilea meniu cu numele **Window**, având opțiunile: **Tile** şi **Cascade**:



5. Adăugați în clasa *Form1* un câmp privat care contorizează numărul de ferestre copil şi o metodă privată care creează o fereastră copil. Pentru aceasta, acționați dublu click pe numele formei în ***Solution Explorer***. În fişierul *Form1.cs*, în clasa `Form1`, scrieți codul:

```
private int nr = 0;     // Numărul de ferestre child

private void CreateChild()
{
    // Creăm o forma child.
     Form f = new Form();
     nr++;

    // Textul din bara de titlu
    f.Text = "Documentul " + nr;

    // Forma f devine document child
    // this este referință la forma container
    f.MdiParent = this;

    // Afişează forma child
    f.Show();
}
```

6. La pornirea aplicației, dorim să se deschidă o fereastră copil. Tratăm evenimentul **Load** generat la încărcarea formei. Dublu click pe suprafața formei. În metoda de tratare a evenimentului, scrieți:

```
private void Form1_Load(object sender, EventArgs e)
```

```
{
    CreateChild();
}
```

7. Acum tratăm evenimentele **Click** pentru toate opțiunile din meniuri. Acționați dublu click pe opțiunea **New**. În metoda de tratare, scrieți:

```
private void newToolStripMenuItem_Click(object sender,
                                        EventArgs e)
{
   // Dacă nu există ferestre copil active
   // setăm contorul la 0
      if (ActiveMdiChild == null)
          nr = 0;
      CreateChild();
}
```

8. Acționați dublu click pe opțiunea **Minimize**. În metoda de tratare, scrieți:

```
private void minimizeToolStripMenuItem_Click(
                           object sender, EventArgs e)
{
     // Dacă există o fereastră copil activă,
     // aceasta se minimizează
     if (ActiveMdiChild != null)
         ActiveMdiChild.WindowState =
                           FormWindowState.Minimized;
}
```

9. Acționați dublu click pe opțiunea **Maximize**. În metoda de tratare, scrieți:

```
private void minimizeToolStripMenuItem_Click(
                           object sender, EventArgs e)
{
     // Dacă există o fereastră copil activă,
     // aceasta se maximizează
     if (ActiveMdiChild != null)
         ActiveMdiChild.WindowState =
                           FormWindowState.Maximized;
}
```

10. Acționați dublu click pe opțiunea **Close**. În metoda de tratare, scrieți:

```
private void closeToolStripMenuItem_Click(
                         object sender, EventArgs e)
{
    // Închide fereastra copil activă
    if (ActiveMdiChild != null)
        ActiveMdiChild.Close();
}
```

11. Acționați dublu click pe opțiunea **Close All** . În metoda de tratare, scrieți:

```
private void closeallToolStripMenuItem_Click(
                            object sender, EventArgs e)
{
    // Închide pe rând fiecare formă copil
    foreach (Form f in this.MdiChildren)
        f.Close();
}
```

12. Acționați dublu click pe opțiunea **Exit.** În metoda de tratare, scrieți:

```
private void exitToolStripMenuItem_Click(object sender,
                                         EventArgs e)
{
    Application.Exit(); // Ieşire din aplicaţie
}
```

13. Acționați dublu click pe opțiunea **Tile** . În metoda de tratare, scrieți:

```
private void tileToolStripMenuItem_Click(object sender,
                                         EventArgs e)
{
    // Aranjare Tile pe orizontală
    this.LayoutMdi(MdiLayout.TileHorizontal);
}
```

14. Acționați dublu click pe opțiunea **Cascade** . În metoda de tratare, scrieți:

```
private void tileToolStripMenuItem_Click(object sender,
                                         EventArgs e)
{
    // Aranjare în cascadă
    this.LayoutMdi(MdiLayout.Cascade);
}
```

15. La închiderea formei, dorim să apară un *message box* cu butoanele **Yes** şi **No**, care să ne întrebe dacă suntem siguri că vrem să ieşim. În *Form Designer* selectați forma container. În fereastra ***Propertie*** apăsați butonul *Events* (fulgerul). Acționați dublu click pe evenimentul **FormClosing**. În metoda de tratare a evenimentului, scriți codul evidențiat în **Bold**:

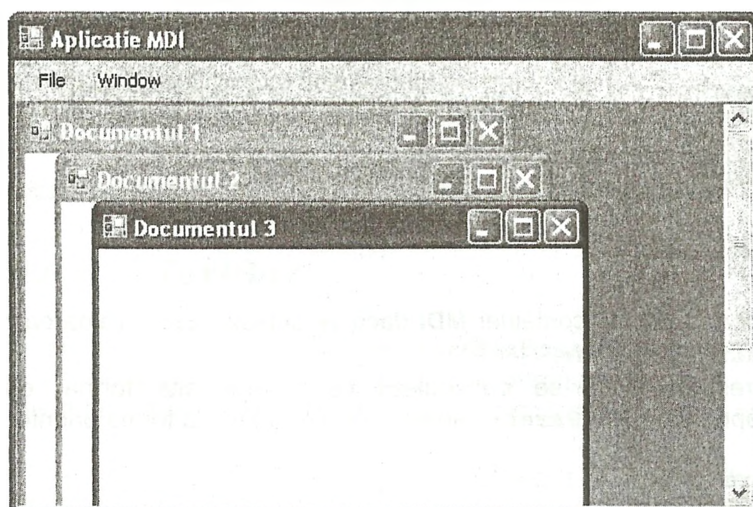
```
private void Form1_FormClosing(object sender,
                              FormClosingEventArgs e)
{
     DialogResult result = MessageBox.Show(
          "Esti sigur ca vrei sa parasesti aplicatia?",
          "Avertizare", MessageBoxButtons.YesNo,
          MessageBoxIcon.Question);
```

```
        // Dacă s-a apăsat No, atunci aplicaţia
        // nu se închide
        if (result == DialogResult.No)
        e.Cancel = true;
    }
```

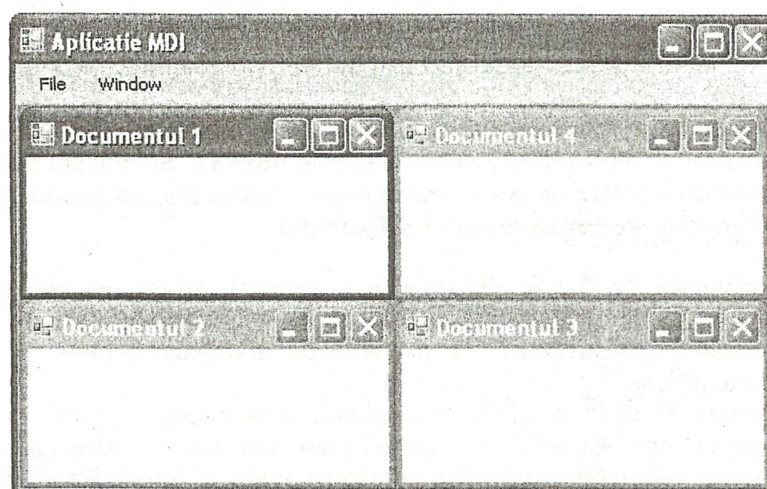
16. Compilaţi şi rulaţi aplicaţia cu **F5**.

Câteva capturi de ecran din timpul rulării:

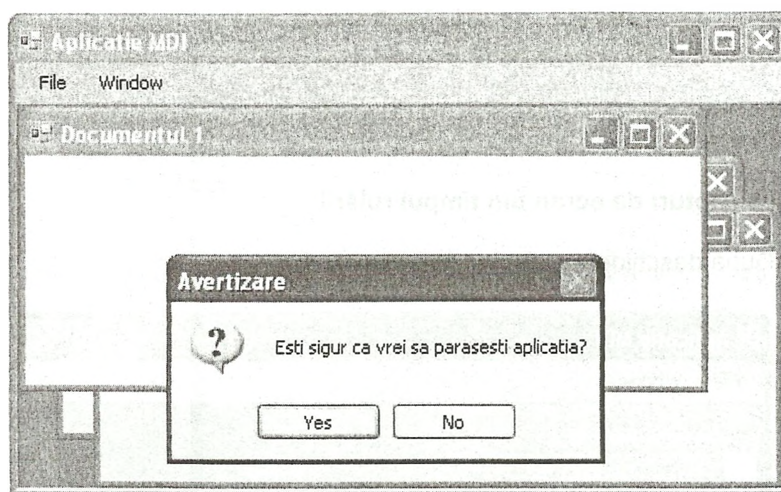
a) După deschiderea câtorva ferestre copil:



b) Aranjare ***Tile*** :



c) La ieşirea din aplicaţie:



De reţinut:

- O formă devine container **MDI** dacă se setează **`true`** valoarea propietăţii **`IsMdiContainer`**.
- Ferestrele *child* se construiesc ca oricare altă formă, după care proprietatăţii **`MdiParent`** i se atribuie o referinţă la forma părinte:

```
Form f = new Form();
f.MdiParent = this;
f.Show();
```

- Proprietatea **`MdiChildren`** a formei container reţine referinţe la toate formele copil.
- Proprietatea **`ActiveMdiChild`** returnează o referinţă la fereastra *child* activă.
- Proprietatea **`WindowState`** a unei forme setează starea curentă a ferestrei. În mod implicit, valoarea ei este **`Normal`**. Se mai pot seta valorile Maximized şi **`Minimized`**. Aceste valori se obţin prin accesarea membrilor enumerării: **`FormWindowState`**. **Exemplu**:

```
ActiveMdiChild.WindowState = FormWindowState.Maximized;
```

- Metoda a **`LayoutMdi()`** a clasei **`Form`**, aranjează formele copil **MDI** în forma părinte.
- Metoda **`Show()`** a clasei **`MessagBox`** este supraîncărcată. Una dintre versiuni, cea din aplicaţie, are trei parametri: textul mesajului, textul din bara de titlu şi butoanele care trebuie să apară pe formă. Reamintim că un *message box* este un dialog predefinit *Windows*, deci o formă, care

returnează întotdeuana o valoare de tip **DialogResult**. Aplicația arată cum puteți exloata valoarea returnată.

- La tratarea evenimentului **FormClosing**, unul dintre parametri este tipul **FormClosingEventArgs**. Clasa are proprietatea **Cancel**. Dacă valorea acestei proprietăți este **true**, atunci evenimentul nu se mai declanşează, deci forma nu se mai închide (scrieți `e.Cancel = true`);

Probleme propuse:

- Implementați aplicației *MdiExample* facilitatea de aranjare a ferestrelor în modul *Tile*, dar pe verticală.
- Adăugați aplicației *MdiExample* un control **ToolStrip**, cu butoane cu aceeaşi funcționalitate cu cea a opțiunilor meniului.
- *Integrați ferestrelor copil în aplicația *MdiExample* un control de tip **TextBox**, pentru ca aplicația să funcționeze ca editor de text.

Controlul RichTextBox

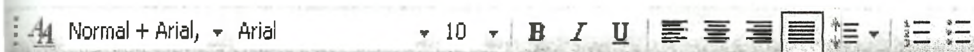
RichTextBox derivă direct din clasa **TextBoxBase**, la fel ca şi **TextBox**. În vreme ce un **TextBox** se utilizează pentru a introduce secvențe scurte de caractere de la tastatură, un **RichTextBox** se utilizează cu precădere ca editor de text.

Controlul are toate facilitățile unui **TextBox**, dar oferă şi altele, cum sunt posibilitatea de formatare a fontului (*Italic*, **Bold**, <u>Underline</u>) şi a paragrafului. Textul se poate introduce de către utilizator, poate fi încărcat din fişiere de tip **RTF** (*Rich Text Format*) sau din fişiere text. Clasa are metode pentru deschidere şi pentru salvare de fişiere în aceste formate. Un control de tip **TextBox** poate fi înlocuit fără modificări majore de cod cu un control **RichTextBox**.

În aplicația care urmează vom utiliza pentru prima oară şi o bară de instrumente, cu ajutorul controlului **ToolStrip**. Înainte de a continua, câteva cuvinte despre acest control:

Controlul ToolStrip

Este un control care vă permite să creați bare de instrumente. Este de fapt un container pentru butoane standard *Windows*. Iată un exemplu:



Se pot crea bare de instrumente cu temă sau fără temă, cu aparența şi comportamentul barelor *Micosoft Office*, sau *Internet Explorer*, dar si bare cu stil şi comportament stabilit de dumneavoastră. Clasa **ToolStrip** abundă de surpize plăcute. Merită să le explorați.

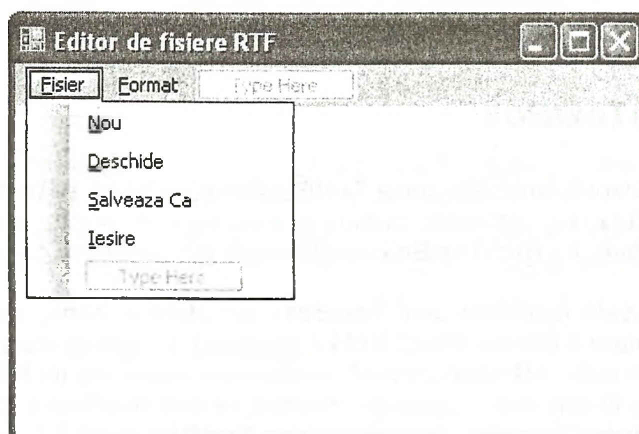
Aplicația *RichTextBoxExample*

Vom realiza un mic editor de text, asemănător celui creat în subcapitolul **TextBox**. Controlul care afişează textul va fi de data aceasta un **RichTextBox**. Aplicația va avea un meniu, facilități de deschidere şi salvare a documentelor în format **RTF**, ca şi posibilitatea de a seta fontul şi culoarea lui. Evident, pentru implementarea acestor facilități, vom utiliza dialoguri predefinite.

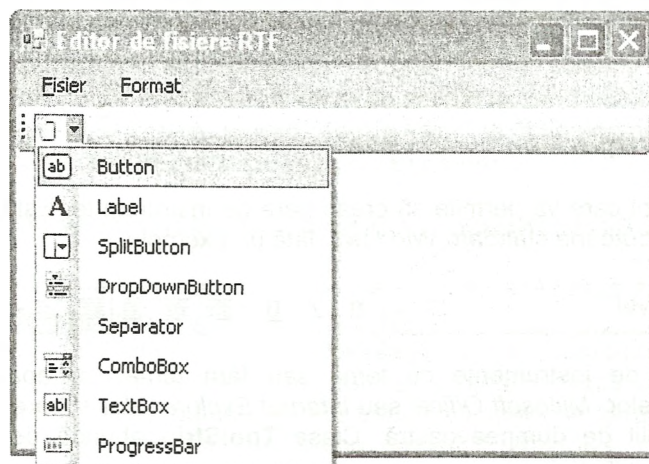
1. Creați un proiect de tip ***Windows Forms Application***, cu numele *RichtextBoxExample*. Selectați forma şi din fereastra ***Properties*** atribuiți proprietății **Text** valoarea: "Editor de fisiere RTF".

2. Din **Toolbox** plasați cu ajutorul mouse-ului un control **MenuStrip** pe suprafața formei.

3. Creați un prim meniu numit **Fisier**, cu opțiunile **Nou**, **Deschide**, **Salveaza Ca**, şi **Iesire**. Creați un al doilea meniu cu numele **Format**, având opțiunile: **Font** şi **Culoare**:



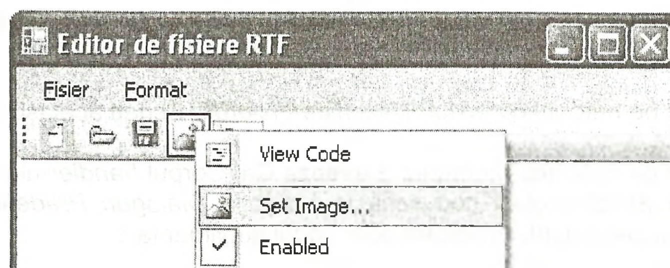
4. Din **Toolbox** plasați pe formă cu ajutorul mouse-ului un control **ToolStrip**.



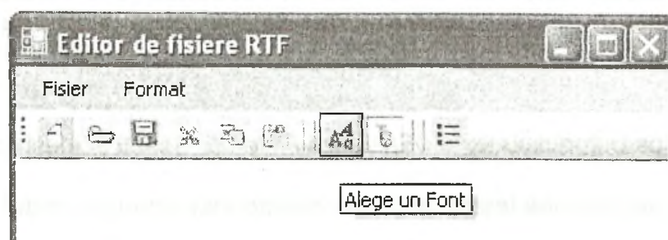
5. În controlul **ToolStrip** adăugați pe rând **9** butoane din lista controlului, așa ca în figura de mai sus. Rolul acestor butoane poate fi descris prin textele: "*Fişier Nou*", "*Deschide Fişierul*", "*Salvează Fişierul*", "*Decupează*", "*Copiază*", "*Lipeşte*", "*Alege un Font*", "*Alege o Culoare*", "*Bullets*". Practic au exact funcționalitatea opțiunilor din meniu.

6. Pe fiecare buton adăugat, vom prefera să lipim câte o imagine, decât să scriem text. Căutați pe Internet câteva imagini **.jpg** sau **.gif** adecvate şi salvați-le într-un folder oarecare. Selectați controlul **ToolStrip** şi din fereastra ***Properties*** setați proprietatea **ImageScalingSize** la valorile **20** şi **20** (implicit e **16** şi **16**). Aceasta, pentru a mări puțin dimensiunile imaginilor afişate pe butoane.

7. Click drept pe fiecare buton din **ToolStrip** şi alegeți **Set Image**. În dialogul **Select Resource**, apăsați butonul **Import**. Veți lipi astfel pe fiecare buton imaginea dorită:



8. Dorim să ataşăm fiecarui buton câte un *Tool Tip Text*, adică un mic text care apare când staționați o clipă cu mouse-ul asupra butonului. Pentru aceasta, selectați pe rând câte un butorn şi din **Properties** atribuiți proprietății **Text** un text scurt, reprezentând rolul butonului:



9. Luați din **Toolbox** un control **RichTextBox** şi plasați-l pe suprafața formei. Acționați click pe sageata din colțul dreapta sus a controlului şi alegeți "*Dock in parent container*".

10. Vom trata evenimentul **Click** pentru fiecare dintre opțiunile meniului, dar şi pentru butoanele din **ToolStrip**. Dublu click pe opțiunea **Nou**. În corpul *handler*-ului de eveniment, scrieți codul:

```
private void nouToolStripMenuItem_Click(object sender,
                                        EventArgs e)
{
     // Şterge textul controlului
     richTextBox1.Text = "";
}
```

Dublu click pe butonul cu textul "*Fisier Nou*" şi scrieţi acelaşi cod.

11. Faceţi dublu click pe opţiunea **Deschide**. Corpul *handler*-ului trebuie să conţină exact acelaşi cod scris în aplicaţia *Dialoguri Predefinite* din subcapitolul precedent. Precizăm doar codul suplimentar:

```
if (openFileDialog1.ShowDialog() ==
                System.Windows.Forms.DialogResult.OK)
{
     // Metoda LoadFile() deschide fişierul
     richTextBox1.LoadFile(openFileDialog1.FileName);
}
```

Dublu click pe butonul cu textul "*Deschide Fisierul*". Scrieţi acelaşi cod.

12. Dublu click pe opţiunea de meniu **Salveaza Ca**. Corpul *handler*-ului trebuie să conţină exact acelaşi cod scris în aplicaţia *Dialoguri Predefinite* din subcapitolul precedent. Precizăm doar codul suplimentar:

```
if (saveFileDialog1.ShowDialog() ==
                System.Windows.Forms.DialogResult.OK)
{
      // Metoda SaveFile() salvează pe disc textul
      // controlului într-un fişier cu formatul RTF
      richTextBox1.SaveFile(saveFileDialog1.FileName,
                 RichTextBoxStreamType.RichText);

}
```

Dublu click pe butonul cu textul "*Salvează Fisierul*". Scrieţi acelaşi cod.

13. Dublu click pe opţiunea **Ieşire**. În corpul *handler*-ului adăugaţi codul:

```
Application.Exit();   // Ieşire din aplicaţie
```

14. Dublu click pe opţiunea de meniu **Font**. *Handler*-ul evenimentului trebuie să arate astfel:

```
private void fontToolStripMenuItem_Click(object sender,
                                         EventArgs e)
{
     if (fontDialog1.ShowDialog() ==
```

```
                    System.Windows.Forms.DialogResult.OK)
    {
        // Porţiunea selectată din text preia
        // caracteristicile alese de către utilizator
        richTextBox1.SelectionFont =
                                    fontDialog1.Font;
    }
}
```

Dublu click pe butonul cu textul "*Alege un Font*". Scrieţi acelaşi cod.

15. Dublu click pe opţiunea de meniu **Culoare**. *Handler*-ul evenimentului trebuie este:

```
private void culoareToolStripMenuItem_Click
                        (objectsender, EventArgs e)
{
    if (colorDialog1.ShowDialog() ==
                System.Windows.Forms.DialogResult.OK)
    {
        richTextBox1.SelectionColor =
                            colorDialog1.Color;
    }
}
```

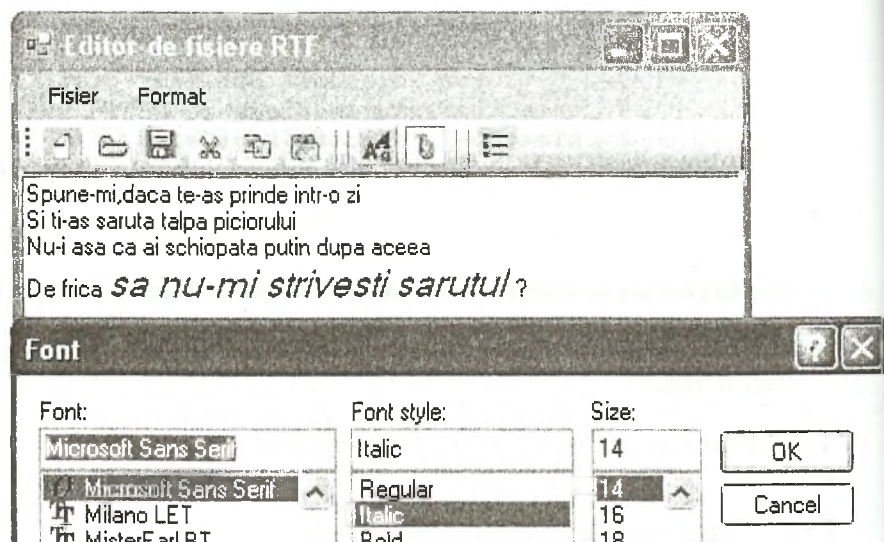
Dublu click pe butonul cu textul "*Alege o Culoare*". Scrieţi acelaşi cod.

16. Selectaţi ultimul buton din **ToolStrip**, cu textul "*Bullets*". În fereastra **Properties**, setaţi proprietatea **Name** la valoarea "*bulletButton*" şi proprietatea **CheckOnClick** la valoarea **true**. Dublu click pe buton pentru a trata evenimentul **Click**. În metoda *handler* scrieţi:

```
private   void   toolStripButton9_Click(object   sender,
EventArgs e)
{
    // Dacă butonul era apăsat atunci textului
    // selectat nu i se aplică stilul bullet
    if ( bulletButton.Checked )
        richTextBox1.SelectionBullet = false;
    else
        richTextBox1.SelectionBullet = true;

}
```

17. Compilaţi şi lansaţi cu **F5**.



Observație:

- Editorul de mai sus lucrează numai cu fişiere **RTF**. La încercarea de deschidere a unui alt tip de fişier, această versiune a metodei `LoadFile()` aruncă o excepție. Din acest motiv, într-o aplicație serioasă, veți apela asemenea metode numai în interiorul blocurilor `try/catch`:

```
try
{
    richTextBox1.LoadFile(openFileDialog1.FileName);
}
catch (System.Exception e)
{
    MessageBox.Show(e.Message);
}
```

  Există o a doua versiune a acestei metode care are doi parametri şi poate fi adaptată la lucrul cu fişiere text.
- Clasa **RichTextBox** are şi alte facilități de editare şi formatare a textului, pe care le puteți explora de exemplu, îmbunătățind editorul de mai sus.

Controlul ToolTip

ToolTip se utilizează pentru afişarea unor mesaje de informare sau de avertizare, atunci când utilizatorul plasează cursorul desupra unui control. O

singură instanță a clasei este suficientă pentru a furniza texte de informare pentru toate controalele unei forme.

Exemplu:

```
ToolTip  t = new ToolTip();
Button   b = new Button();
ComboBox c = new ComboBox();
t.SetToolTip(b, "Valideaza datele");
t.SetToolTip(c, "Lista elevilor din clasa a XII-a B");
```

Dacă doriți stiluri şi afişări diferite pe aceeaşi formă, veți utiliza mai multe controale **ToolTip**.

Principalii membri ai clasei ToolTip

Proprietăți :

`AutoPopDelay`	- Returnează sau setează perioada de timp în care **ToolTip** rămâne vizibil dacă cursorul e staționar deasupra controlului.
`IsBallon`	- Returnează o valoare de tip **bool** care indică dacă mesajul este inlclus într-o fereastră balon.
`ToolTipIcon`	- Setează un icon pentru **ToolTip**.
`ToolTipTitle`	- Stabileşte un titlu pentru **ToolTip**

Metode:

`Hide()`	- Ascunde fereastra **ToolTip**
`Show()`	- Setează un text pentru **ToolTip** şi îl afişează
`RemoveAll()`	- Înlătură toate textele asociate cu un control **ToolTip**

Evenimente:

`Popup`	- Se declanşează înainte ca **ToolTip** să fie afişat

Aplicația *ToolTipExample*

În proiectul de față se foloseşte acelaşi **ToolTip** pentru mai multe controale de pe aceeaşi formă. Se utilizează *designer*-ul pentru toate acțiunile necesare.

1. Creați un proiect de tip ***Windows Forms Application*** cu numele *ToolTipExample*.

2. Din **Toolbox** plasați pe formă câteva controale: două controale de tip **TextBox**, un **GroupBox**, două controale de tip **RadioButton**, un **CheckBox** şi două butoane:

3. Din ToolBox alegeți un **ToolTip** şi plasați-l pe suprafața formei. Acesta nu este vizibil pe formă, dar referința `toolTip1` apare în *designer tray*.

4. Selectați primul control **TextBox**. În fereastra **Properties** selectați *ToolTip on toolTip1*. Acționați click pe săgeata din dreapta şi în panoul care se deschide introduceți mesajul: *"Introduceți numele, inițiala tatălui şi prenumele"*. Apoi acționați **Ctrl + Enter**.

5. Selectați al doilea control **TextBox**. Acționați de aceeaşi manieră ca la pasul **4**, dar introduceți alt mesaj, cum ar fi *"Introduceți strada, numărul locuinței şi localitatea"*.

6. Se repetă pasul **4**, pentru controalele de tip **RadioButton**, pentru **CheckBox** şi pentru cele două controale de tip **Button**.

7. Compilați şi rulați cu **F5**.

La rulare, când staționați cu mouse-ul asupra fiecărui control, apar mesajele **ToolTip**:

De reținut:

- Mesajele **ToolTip** se pot seta atât *design time*, cât şi *run time*.
- Se poate introduce **ToolTip** fără ajutorul *designer*-ului. Tratați de exemplu evenimentul **Load** pentru forma acestei aplicații (dublu click pe suprafața formei) şi introduceți codul:

```
private void Form1_Load(object sender, EventArgs e)
{
    ToolTip t = new ToolTip();
    t.SetToolTip(textBox2,
       "Introduceți numele, inițiala tatălui şi prenumele");
    t.SetToolTip(textBox1,
             "Introduceți strada şi numărul locuinței");
    t.SetToolTip(radioButton1,
             "Studiile postliceale sunt considerate medii");
    t.SetToolTip(radioButton2,
             "Licență, masterat sau doctorat");
    t.SetToolTip(checkBox1,
             "Nu bifați în cazul în care sunteți divorțat!");
    t.SetToolTip(button1,
             "Se validează datele introduse");
    t.SetToolTip(button2,
             "Se renunță la datele introduse");
}
```

Controlul NotifyIcon

Formele, ca de altfel şi alte controale, moştenesc metoda `Hide()` de la clasa `Control`. Dacă utilizatorul ascunde forma principală a aplicației, atunci nu mai are posibilitatea să o reafişeze.

Clasa `NotifyIcon` oferă posibilitatea de a accesa procese care rulează în *background*. Creează un icon în zona de notificare (*notify area*) din *system tray*. Zona de notificare se găseşte în dreapta *task bar*-ului, lângă ceas. Iconul răspunde la evenimente, cum sunt **Click** sau **DoubleClick** pe care le puteți trata pentru a reafişa fereastra aplicației. Alternativ, se obişnuieşte să se ataşeze un meniu contextual care se deschide la click drept pe icon.

Principalii membri ai clasei

Proprietăți :

`ContextMenu`	-	Returnează sau setează un meniu de context pentru icon
`Icon`	-	Returnează sau setează un iconul controlului
`Text`	-	Returnează sau setează un text *ToolTip*. cînd cursorul mouse-ului staționează deasupra iconului
`Visible`	-	Stabileşte dacă iconul e vizibil în zona de notificare

Metode:

`ShowBallonTip()`	-	Afişează mesajul **ToolTip** într-o formă asemănătoare unui balon

Evenimente:

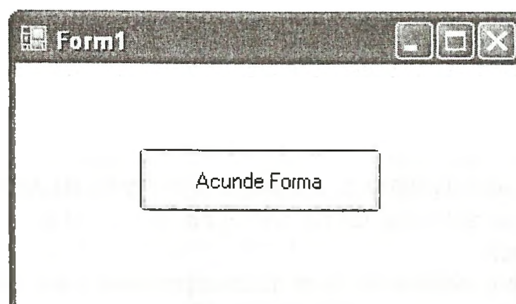
`Click`	-	Se declanşează la click pe icon
`DoubleClick`	-	Se declanşează la dublu click pe icon

Aplicaţia *NotifyIconExample*

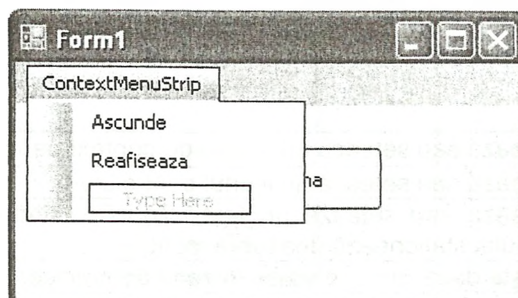
Pe o formă plasăm un buton care la click o ascunde. Folosim un **NotifyIcon** pentru reafişarea formei. Reafişarea se produce la dublu click pe icon sau la click într-un meniu de context.

Urmaţi paşii:

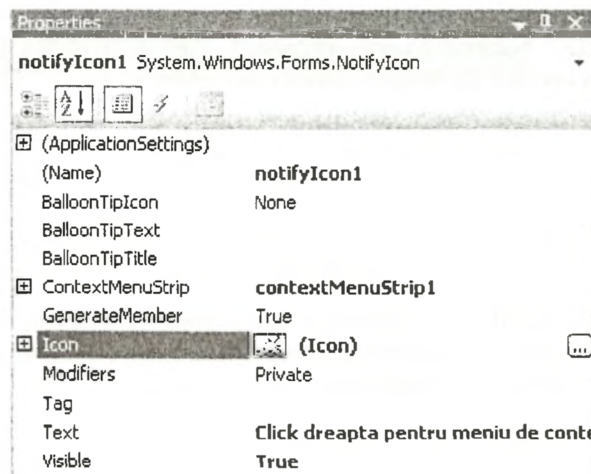
1. Creaţi un proiect de tip ***Windows Forms Application*** cu numele *NotifyIconExample*.
2. Din **Toolbox** plasaţi pe formă un control de tip **Button**. În fereastra **Properties** setaţi proprietea `Text` la valoarea "*Ascunde Forma*":



3. Trageţi deasupra formei un control de tip **ContextMenuStrip**. Selectaţi în *designer tray* referinţa `contextMenuStrip1`. Introduceţi itemii *Acunde* şi *Reafiseaza*:



4. Trageți deasupra formei un control de tip **NotifyIcon**. Selectați in *designer tray* referința `notifyIcon1`.
   a. În fereastra **Properties**, selectați proprietatea `Text` şi introduceți textul care doriți să apară ca *tool tip* când staționați cu mouse-ul deasupra iconului.
   b. Setați proprietatea `Visible` la `true`.
   c. Selectați proprietatea `Icon` şi alegeți iconul care apare în zona de notificare. Evident, trebuie să aveți unul pregătit. Nu uitați să setați un icon. În lipsa lui, nu apare la execuție nimic în *notify area*.
   d. Selectați proprietatea `ContextMenuStrip` şi alegeți valoarea *contextMenuStrip1*.



5. Tratăm evenimentul **Click** pentru butonul *Ascunde Forma*. Executați dublu click pe el şi scrieți în corpul *handler*-ului codul evidențiat în **Bold**:

```
private void button1_Click(object sender, EventArgs e)
{
    // Ascunde forma aplicaţiei
    this.Hide();
}
```

6. În *designer tray* selectați referința `notifyIcon1` şi faceți dublu click pe opțiunea *Ascunde*, a meniului contextual. În *handler*-ul de tratare a evenimentului **Click**, introduceți codul:

```
private void ascundeToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
    // Ascunde forma aplicaţiei
    this.Hide();
}
```

7. Acționați dublu click pe opțiunea *Reafiseaza* a meniului contextual, pentru tratarea evenimentului **Click**. În corpul metodei de tratare, introduceți codul:

```
private void reafiseazaToolStripMenuItem_Click(
                          object sender, EventArgs e)
{
    this.Visible = true;
}
```

8. Dorim ca la dublu click pe icon în *notify area* forma să redevină vizibilă. Pentru aceasta, tratăm evenimentul **DoubleClick** pentru **NotifyIcon**. Selectați în *designer tray* referința **notifyIcon1**, apoi în fereastra **Properties** selectați butonul *Events* (fulgerul). Acționați dublu click pe evenimentul **DoubleClick**. În corpul metodei de tratare scrieți:

```
private void notifyIcon1_DoubleClick(object sender,
                                     EventArgs e)
{
    this.Visible = true;
}
```

9. Tratăm evenimentul **Click** pentru opțiunile din meniul contextual. Selectați în *designer tray* referința **contextMenuStrip1**. Acționați dublu click pe opțiunea *Ascunde*. În editor, scrieți codul:

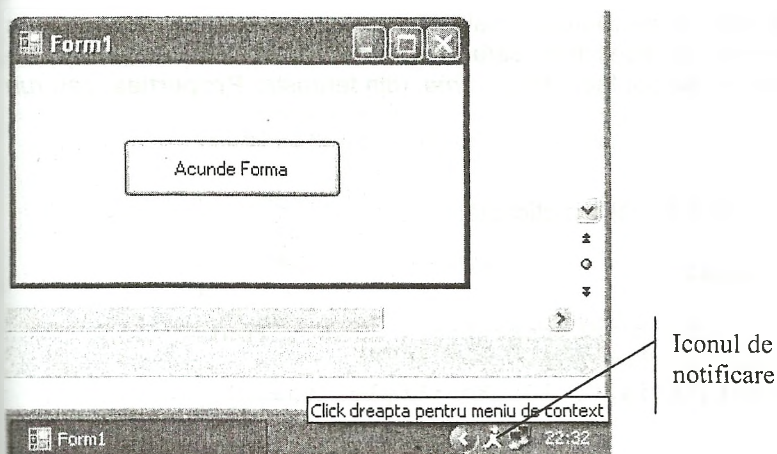
```
private void ascundeToolStripMenuItem_Click(
                         object sender, EventArgs e)
{
    this.Hide();
}
```

10. Acționați dublu click pe opțiunea *Reafiseaza*. În editor scrieți codul:

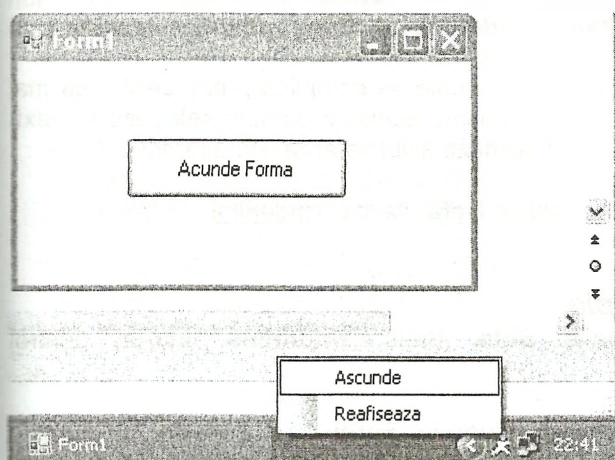
```
private void reafiseazaToolStripMenuItem_Click(
                         object sender, EventArgs e)
{
    this.Visible = true;
}
```

11. Compilați și rulați cu **F5**.

La executare, iconul rămâne în permanență vizibil în zona de notificare.



La click pe butonul *Ascunde Forma*, forma devine invizibilă. La dublu click pe icon, forma redevine vizibilă, iar iconul rămâne vizibil. Din meniul de context puteți de asemenea să controlați vizibilitatea formei:



De reținut:

- Controlul **NotifyIcon** plasează un icon în zona de notificare din *system tray*. Iconul reprezintă o scurtătură pentru procesele care lucrează în *background*.
- Controlul răspunde la evenimente (**Click**, **DoubleClick**) şi admite un meniu contextual.

Fonturi

Fonturile nu sunt controale **.NET**. Clasa **Font** este definită în spațiul de nume **System.Drawing**. Un obiect de tip **Font** încapsulează un format particular

de text, incluzând stilul, dimensiunea. Toate controalele de bază au proprietatea **Font**, care vă permite să specificați caracteristicile fontului care va fi afişat în control. Setările fontului se pot face *design time*, (din fereastra **Properties**) sau *run time*.

Exemplu:

Setăm fontul afişat de către o etichetă:

```
Label e = new Label();

// Utilizăm unul dintre constructorii clasei Font
// pentru a crea un nou font
e.Font = new Font("Arial", 10, FontStyle.Italic);
```

Stilurile Fonturilor

Enumerarea **FontStyle** din spațiul de nume **System.Drawing** are ca membri enumeratorii: **Regular**, **Bold**, **Italic**, **Underline** şi **Strikeout**. Aceştia determină stilul aplicat textului.

Cînd lucrați cu un editor de text, lucrurile se complică puțin, pentru că mai multe stiluri din cele de mai sus pot fi aplicate aceleaşi porțiuni selectate de text. Aplicarea unui nou stil nu trebuie să invalideze stilul anterior. De exemplu:

"Această ***<u>portiune de text</u>*** conține stilurile **Bold**, *Italic* şi <u>Underline</u>".

Aplicația *FontStyleExample*

Proiectul introduce stilurile **Bold**, **Italic**, **Underline**, într-un control **RichTextBox**:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *FontStyleExample*.

2. Din **Toolbox** alegeți un control **TooStrip** şi plasați-l pe suprafața formei. Adăugați trei butoane în acest control. Selectați pe rând cele trei butoane şi din fereastra **Properties** atribuiți proprietății **Text** valorile *Bold*, *Italic*, *Underline*, iar proprietății **DysplayStyle** valoarea *Text*.

3. Aduceți pe suprafața formei un control de tip **RichTextBox**. Setați din fereastra **Properties**, câmpul **Dock** la valoarea *Fill*.

4. Tratăm evenimentul **Click** pentru fiecare buton. Dublu click pe butonul **Bold**. *Handler*-ul de eveniment este:

```
private void toolStripButton1_Click(object sender,
                                    EventArgs e)
```

```
{
    Font vechiulFont, noulFont;

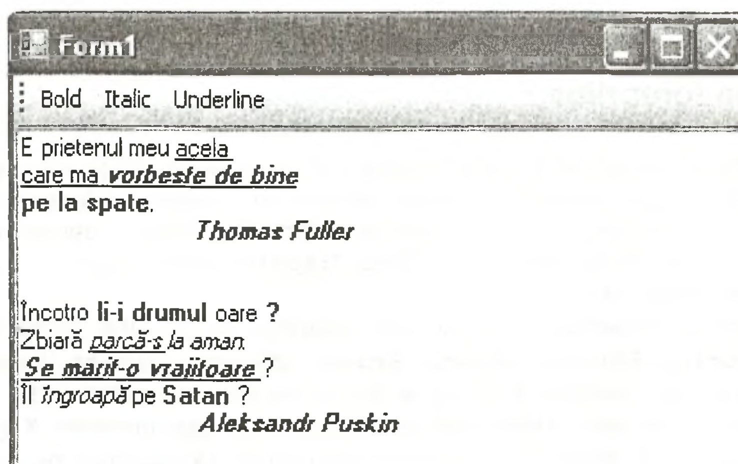
    // Returnează fontul folosit în textul selectet
    vechiulFont = this.richTextBox1.SelectionFont;

    // Dacă vechiul font era stil Bold, înlăturăm
    // formatarea
    if (vechiulFont.Bold)
    {
        noulFont = new Font(vechiulFont,
              vechiulFont.Style & ~ FontStyle.Bold);
    }
    else
    {
        noulFont = new Font(vechiulFont,
               vechiulFont.Style | FontStyle.Bold);
    }
    // Inserăm noul font si redăm focusul
    // controlului RichTextBox.
    this.richTextBox1.SelectionFont = noulFont;
    this.richTextBox1.Focus();
}
```

5. Dublu click pe butonul **Italic**. *Handler*-ul de eveniment va avea acelaşi cod cu cel ataşat evenimentului **Bold**, cu unica diferenţă că în toate locurile cuvântul "`Bold`" se înlocuieşte cu "`Italic`".

6. Dublu click pe butonul **Underline**. *Handler*-ul de eveniment va avea acelaşi cod cu cel ataşat evenimentului **Bold**, cu unica diferenţă că în toate locurile cuvântul "`Bold`" se înlocuieşte cu "`Underline`".

7. Compilaţi şi rulaţi cu **F5**.



Observație:

- Codul prezent în metodele *handler* verifică mai întâi dacă stilul dorit este deja prezent. Dacă este, îl înlătură.
- Pentru adăugarea sau înlăturtarea unui stil nou, fără a afecta celelalte stiluri prezente, se efectuează operații pe biți cu enumerarea **FontStyle**. ("*Or logic pe biți*" (|) adaugă un stil nou, iar "*And logic pe biți*", aplicat negării noului stil, înlătură noul stilul dintre cele prezente. Exemple:

```
// Adaugă stilul Underline:
vechiulFont.Style | FontStyle.Underline

// Înlătură stilul Underline
vechiulFont.Style & ~ FontStyle.Underline
```

Fonturile instalate

Cu ajutorul unui obiect al clasei `InstalledFontCollection`, definită în spațiul de nume **System.Drawing.Text** se obțin fonturile instalate în sistem, astfel:

```
using System.Drawing.Text;
InstalledFontCollection fnt = new InstalledFontCollection();
```

Proprietatea `Families` a clasei `InstalledFontCollection` returnează un tablou de referințe la obiecte de tip `FontFamily`. Fiecare obiect încapsulează o familie de fonturi, iar prin proprietatea **Name**, returnează numele fontului:

```
foreach (FontFamily fam in fnt.Families)
    System.Console.WriteLine(fam.Name);
```

Vom vedea un exemplu după paragraful următor.

Desenarea fonturilor

.NET Framework oferă posiblitatea de a desena pe suprafețele formelor sau ale controalelor. Se pot desena linii, forme geometrice, imagini sau fonturi. În acest paragraf ne vom ocupa numai de desenarea fonturilor. Pentru desenare, aveți nevoie de un obiect de tip `Graphics`. Clasa `Graphics` este definită în spațiul de nume `System.Drawing`.

Fonturile se desenează cu metoda `DrawString()`. Una dintre versiuni este: `DrawString(String, Font, Brush, float, float)`. Desenează textul specificat de obiectele `String` şi `Font`, cu pensula indicată de obiectul `Brush`, la locația indicată. Ultimii doi parametrii indică coordonatele **X** şi **Y** ale colțului stânga sus al textului. Dacă de exemplu dorim să desenăm pe suprafața

unei forme, atunci cel mai bun moment pentru desenare este acela al desenării formei şi vom trata evenimentul `Paint` al formei.

Aplicaţia *FontDrawingExample*

Proiectul de faţă determină fonturile instalate în sistem şi populează un combobox cu numele acestora. La selectarea unui font din listă, numele fontului se va desenează pe suprafaţa formei.

1. Creaţi un nou proiect de tip ***Windows Forms Application***, cu numele *FontDrawingExample*.

2. Din Toolbox, plasaţi pe suprafaţa formei un control de tip **ComboBox**.

3. Tratăm evenimentul `Load` generat la încărcarea formei. Acţionaţi dublu click pe suprafaţa formei. Scrieţi codul marcat în **Bold**, în *handler*-ul de eveniment:

```
private void Form1_Load(object sender, EventArgs e)
{
      // Determină fonturile instalate
      InstalledFontCollection fnt =
                              new InstalledFontCollection();

      // Adaugă numele fiecărui font în listă
      foreach (FontFamily fam in fnt.Families)
          comboFont.Items.Add(fam.Name);
}
```

4. În antetul fişierului `Form1.cs`, adăugaţi directiva:

```
using System.Drawing.Text;
```

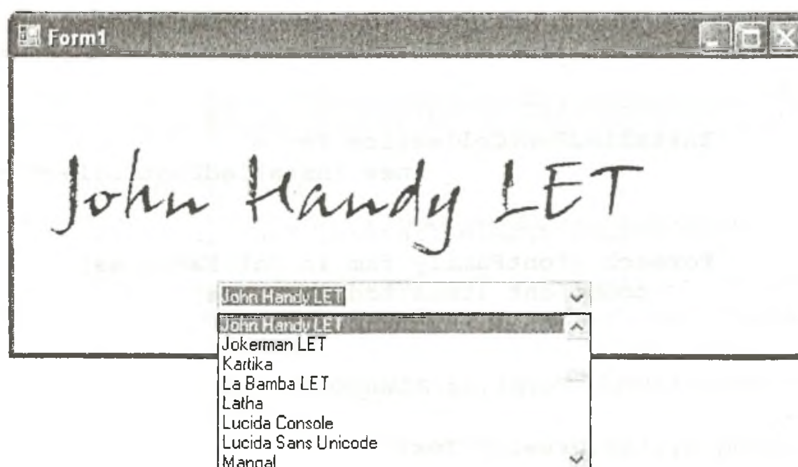
5. Dorim ca desenarea fontului să se producă odată cu redesenarea formei. Forţăm redesenarea formei, la fiecare selectare a unui item în **ComboBox**. Tratăm evenimentul **SelectedIndexChanged**. În acest scop, faceţi dublu click pe controlul **ComboBox**. Scrieţi codul:

```
private void comboBox1_SelectedIndexChanged
                              (object sender, EventArgs e)
{
      // Invalidează suprafaţa formei, fapt
      // care cauzează redesenarea acesteia
      this.Invalidate();
}
```

6. Tratăm evenimentul **Paint** pentru formă. Selectaţi forma în *Form Designer* şi din fereastra **Properties** acţionaţi dublu click pe evenimentul `Paint`. Scrieţi codul marcat în **Bold**, în *handler*-ul de eveniment:

```
private void Form1_Paint(object sender,
                         PaintEventArgs e)
{
    try
    {   // Desenează fontul pe suprafaţa formei
        e.Graphics.DrawString(comboBox1.Text,
                     new Font(comboBox1.Text, 40),
                     Brushes.Black, 10, 50);
    }
    catch (ArgumentException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

7. Compilaţi şi lansaţi în execuţie cu **F5**.



TabControl

Un **TabControl** conţine pagini suprapuse, numite pagini *tab*. O pagină *tab* este o instanţă a clasei **TabPage**. Pagina *tab* curentă se schimbă prin cu click cu mouse-ul pe *tab*-ul dorit sau se poate face programatic. Paginile *tab* suportă alte controale, în mod similar formelor. Astfel, un **TabControl** poate constitui o alternativă unei afişări succesive a mai multor forme.

Principalii membrii ai clasei TabControl

Proprietăţi :

`SelectedIndex`	- Returnează sau setează indexul paginii *tab* selectate
`SelectedTab`	- Returnează sau setează pagina *tab* selectată
`TabCount`	- Returnează numărul de *tab*-uri din control
`TabPages`	- Returnează colecția de *tab*-uri din **TabControl**

Metode:

`GetControl()`	- Returnează obiectul **TabPage** de la locația specificată
`GetItems()`	- Returnează colecția de obiecte **TabPage** aparținând controlului **TabControl**.

Evenimente:

`Selected`	- Se declanşează când se selectează un *Tab*
`SelectedIndexChanged`	- Se declanşează când proprietatea `SelectedIndex` s-a modificat

Aplicația *TabControlExample*

Aplicația pe care o propunem utilizează un **TabControl** pentru afişarea imaginilor. *Tab*-urile controlului se adaugă *runtime*, în momentul în care se încarcă o imagine de pe disc. Operațiile se execută la apăsarea butoanelor unui control **ToolStrip**. Acestea sunt:

- **Open** – încarcă o imaginede pe disc şi o afişează într-un tab nou.
- **Next** – selectarea *tab*-ului următor
- **Prev** - selectarea *tab*-ului anterior
- **Close** – închiderea *tab*-ului curent
- **Close All** – închiderea tuturor *tab*-urilor

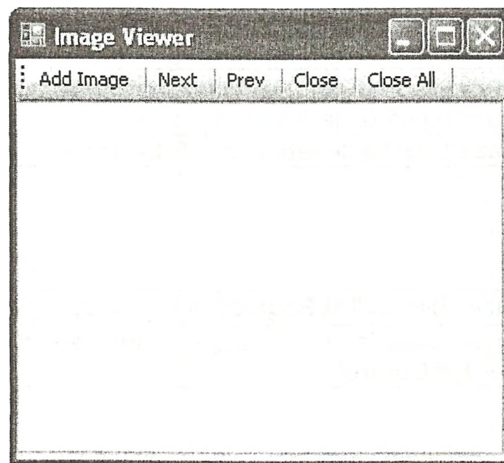
Urmați paşii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *TabControlExample*.

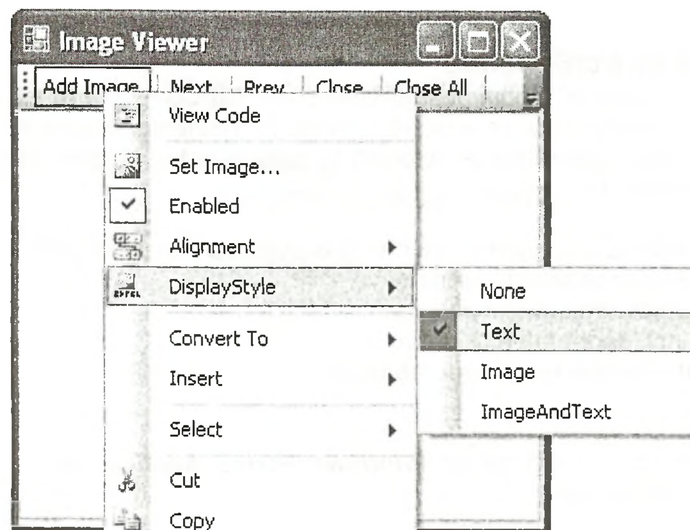
2. Din Toolbox, plasați pe suprafața formei un control de tip **ToolStrip** şi un **TabControl**.

3. Selectați **TabControl**-ul, apoi în fereastra **Properties** setați proprietatea **`Dock`** la valoarea **`Fill`**. Ştergeți cele două tab-uri implicite din control (acționați click dreapta pe fiecare *tab* şi selectați) *Remove Tab*.

4. Adăugați cinci butoane pe **ToolStrip**, despărțite printr-un obiect de tip separator. Forma trebuie să arate astfel:



Butoanele din **ToolStrip** trebuie să afişeze text. Ca să obţineţi asta, acţionaţi click dreapta pe fiecare buton, apoi alegeţi *DisplayStyle* şi selectaţi *Text*. Mai departe, în fereastra **Properties**, atribuiţi proprietăţii **Text** valorile corespunzătoare:



5. Declarăm un câmp privat în clasa `Form1`, care stabileşte caracterul separator pentru căile de director. În ***Solution Explorer***, click dreapta pe *Form1*, selectaţi *View Code*, apoi în fişierul *Form1.cs*, în clasa `Form1`, scrieţi:

```
private char[] sep = new char[] { '\\' };
```

6. Din **Toolbox**, trageţi pe suprafaţa formei un control de tip **OpenFileDialog**.

7. Tratăm evenimentul **Click** pentru butonul *Add Image*. Acționați dublu click pe buton. În *handler*-ul de eveniment, scrieți codul evidențiat în **Bold**:

```
private void toolStripButton1_Click(object sender,
                                    EventArgs e)
{
    string[] aux;
    string fullPath; // Calea până la fişierul imagine
    string imgName;  // Numele fişierului imagine

    // Setăm un filtru pentru fişiere de tip imagine
    openFileDialog1.Filter = "Fisiere suportate " +
        " (*.jpg;*.png;*.ico;*.gif;*.bmp;*.tiff)" +
        "|*.jpg;*.png;*.ico;*.gif;*.bmp;*.tiff "  +
        "|All files (*.*)|*.*";

    // Dacă dialogul se închide prin apăsarea OK
    if (openFileDialog1.ShowDialog() ==
        DialogResult.OK)
    {
        // Obţinem calea până la fişier
        fullPath = openFileDialog1.FileName;

        // Separăm calea în stringuri separate prin '\'
        aux = fullPath.Split(sep);

        // Numele imaginii e ultimul string
        imgName = aux[aux.Length - 1];

        // Creăm un tab nou şi un PictureBox
        TabPage tP = new TabPage(imgName);
        PictureBox pB = new PictureBox();

        // Încărcăm în picture box imaginea
        pB.Image = Image.FromFile(fullPath);

        // Stabilim poziţia şi dimensiunea imaginii pe
        // tab. (x şi y faţă de colţul stânga sus a
        // tab-ului, lăţimea şi înălţimea)

        pB.SetBounds(
          (tabControl1.Width/2) - pB.Image.Width / 2),
          (tabControl1.Height/2)-(pB.Image.Height/2),
           pB.Image.Width, pB.Image.Height);

        // Adăugăm pb pe tab
        tP.Controls.Add(pB);

        // Adăugăm tab-ul pe TabControl
        tabControl1.TabPages.Add(tP);
```

```
            // Tab-ul adăugat devine cel selectat
            tabControl1.SelectedIndex =
                  tabControl1.TabPages.Count - 1;
        }
    }
```

8. Tratăm evenimentul **Click** pentru butonul *Next*. Faceți dublu click pe buton. În metoda de tratare, scrieți:

```
private void toolStripButton2_Click(object sender,
                                    EventArgs e)
{
            // Dacă există tab-ul următor
        if (tabControl1.SelectedIndex + 1 <
                          tabControl1.TabPages.Count)
        {
            // îl selectăm
            tabControl1.SelectedIndex =
                        tabControl1.SelectedIndex + 1;
        }
}
```

9. Tratăm evenimentul **Click** pentru butonul *Prev*. Faceți dublu click pe buton. În metoda de tratare, scrieți:

```
private void toolStripButton3_Click(object sender,
                                    EventArgs e)
{
            // Dacă există tab-ul precedent
        if (tabControl1.SelectedIndex - 1 >= 0)
        {
            // îl selectăm
            tabControl1.SelectedIndex =
                        tabControl1.SelectedIndex - 1;
        }
}
```

10. Tratăm evenimentul **Click** pentru butonul *Close*. Faceți dublu click pe buton. În metoda de tratare, scrieți codul evidențiat în **Bold**:

```
private void toolStripButton4_Click(object sender,
                                    EventArgs e)
{
     // Dacă mai există pagini (tab-uri)
     if (tabControl1.TabPages.Count > 0)
     {
        // O ştergem pe cea selectată
        tabControl1.TabPages.RemoveAt(
                          tabControl1.SelectedIndex);
     }
}
```

11. Tratăm evenimentul **Click** pentru butonul *Close All*. Faceți dublu click pe buton. În metoda de tratare, scrieți codul evidențiat în **Bold**:

```
private void toolStripButton5_Click(object sender,
                                    EventArgs e)
{
    // Cât timp mai sunt pagini (tab-uri)
    while (tabControl1.TabPages.Count > 0)
    {
        // ştergem pagina selectată
        tabControl1.TabPages.RemoveAt(
                         tabControl1.SelectedIndex);
    }
}
```

12. Compilați şi rulați cu **F5**.

La rulare, cu *Add Image* se pot încărca mai multe imagini în pagini diferite, cu butoanele *Next* şi *Prev* se parcurg tag-urile deschise, cu *Close* se închide *tab*-ul selectat, iar cu *Close All* se închid toate paginile.

Probleme propuse

1. Realizați o aplicație care utilizează un **TabControl** care preia comenzile clienților la o pizzerie. Paginile controlului trebuie să conțină controale de tip **TextBox**, **ComboBox**, **CheckBox**, **RadioButton** şi **Button**. La finalul comenzii, aplicația afişează într-un dialog întreaga comanda, împreună cu prețul final.

2. Realizați un proiect care un utilizează un **TabControl** cu şapte pagini. Controlul reprezintă jurnalul unui elev şi fiecare pagină corespunde unei zile a săptămânii. Controlul trebuie să conțină cel puțin căsuțe de editare. După completarea zilei a şaptea, se apasă un buton care creează un nou *tab*. În acest *tab* se vor afişa într-un **TextBox** informațiile introduse în timpul săptămânii (în rezumat sau integral - dumneavoastră decideți).

Controalele ListBox, ComboBox şi CheckedListBox

Toate cele trei controale moştenesc clasa **ListControl** şi servesc la afişarea unei liste de itemuri. Itemurile apar în listă ca şiruri de caractere şi pot fi selectate prin click cu mouse-ul.

Controlul **ListBox** poate oferi selecție simplă sau multiplă, în funcție de valoarea proprietății **`SelectionMode`**.

Controlul **ComboBox** combină un **TextBox** cu un **ListBox**, permițând utilizatorului să selecteze itemi dintr-o listă, sau să introducă noi itemi.

Controlul **CheckedListBox** afişează o listă de itemi care pot selectați cu ajutorul unui checkbox prezent pentru fiecare item.

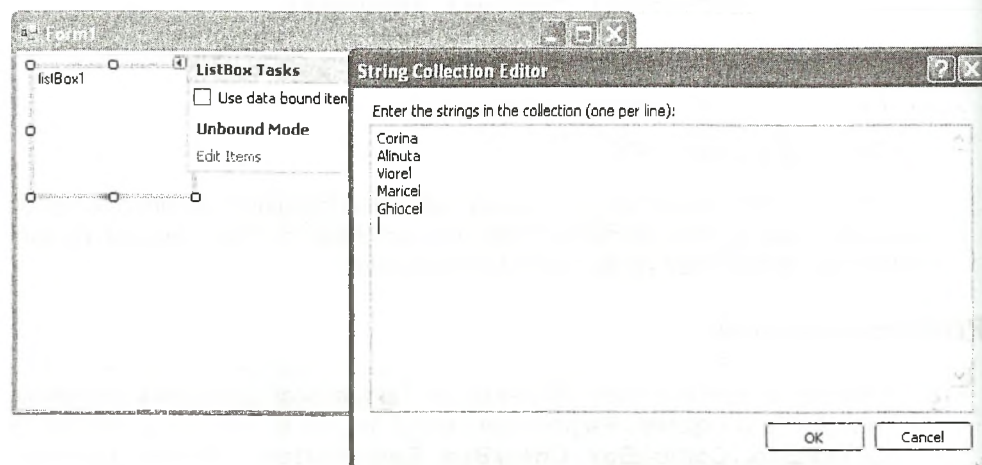
Reţineţi că în aceste tipuri de liste se pot însera obiecte, dar că ceea ce se vede este reprezentarea ca şir de caractere a acestor obiecte.

Adăugarea itemilor în liste

Itemii se pot adăuga *design time* sau *run time*.

Adăugarea itemilor *design time*

O variantă simplă este să selectaţi controlul, faceţi click pe săgeata din colţul stânga sus, alegeţi *Edit Items* şi introduceţi itemii manual, câte unul pe linie:



Adăugarea itemilor *run time*

Se folosesc metodele `Insert()`, `Add()` sau `AddRange()`. Cu `Insert()` inseraţi un item la o poziţie specificată. De exemplu:

```
// Înserează în listă la poziţia 3 un item cu textul "Item 3"
listBox.Items.Insert(3, "Item 3");
```

Metoda `Add()` adaugă la sfârşitul listei un obiect.
Exemplu:

```
comboBox.Items.Add("Item Nou");
```

Pentru adăugarea eficientă a unui număr mare de itemi în listă, controalele furnizează metodele `BeginUpdate()` şi `EndUpdate()`. Prin utilizarea acestora, controlul nu este redesenat de fiecare dată când un item este introdus în listă.
Exemplu:

```
comboBox.BeginUpdate();    // Controlul nu se mai redeseneaza
for (int i = 0; i < 200; i++)
    comboBox.Items.Add("Itemul " + i);
comboBox.EndUpdate();
```

Adaugarea unui interval de itemuri în liste se face cu metoda `AddRange()`. Aceasta primeşte un tablou de obiecte, apoi afişează în listă valoarea de tip `string` implicită pentru fiecare obiect.
Exemplu:

```
listBox.Items.AddRange(new object[] { "UNU", "DOI", "TREI"});
```

Listele pot prelua itemi. Sursele de date pot fi tablouri, colecţii, *DataSet*-uri.
Exemplu:

```
string[] copii = { "Ionel", "Viorel", "Alinel", "Dorel" };
checkedListBox.DataSource = copii;
```

Itemurile se şterg cu metoda **Remove()**. **Exemplu**:

```
// Şterge obiectul caracterizat de şirul "Alin"
listBox.Items.Remove("Alin");

// Şterge primul item
lisBox.Items.RemoveAt(0);
```

Clasele specifice acestor liste au desigur şi alte metode şi proprietăţi utile. Vom face un mic proiect, pentru introducere în lucrul cu controalele de tip listă.

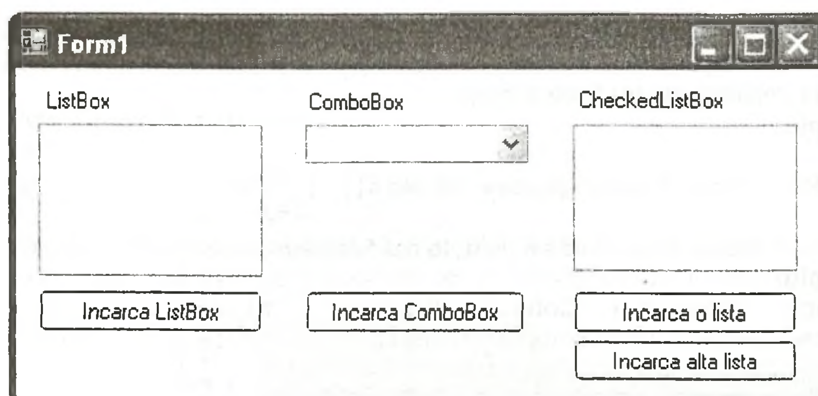
Aplicaţia *TestListControl*

Aplicaţia pe care o propunem ilustrează diferite modalităţi de populare *runtime* a controalelor **ListBox, ComboBox** şi **CheckedListBox**:

1. Creaţi un nou proiect de tip ***Windows Forms Application***, cu numele *TestListControl*.

2. Din **Toolbox** plasaţi pe formă cu ajutorul mouse-ului un control **ListBox**, un control **ComboBox** şi un control **CheckedListBox**. Adăugaţi de asemenea trei **Label**-uri pe care le veţi eticheta corespunzător controalelor de tip listă.

3. Sub primele două controale de tip listă plasaţi câte un buton. Sub controlul de tip `CheckListBox` plasaţi două butoane. Stabiliţi valorile proprietăţilor **Name** ale butoanelor astfel: `listButton`, `comboButton`, `checkListButton1` şi `checkListButton2`, iar valorile proprietăţilor **Text**, aşa cum se vede în figură:



4. Dorim ca la încărcarea formei să încărcăm şi controlul **CheckListBox** cu o primă listă de itemi. Pentru aceasta, tratăm evenimentul **Load** care se declanşează la încărcarea formei. Faceţi dublu click pe suprafaţa formei. Completaţi astfel corpul *handler*-ului de eveniment:

```
private void Form1_Load(object sender, EventArgs e)
{
      // Un tablou de referinţe la obiecte de tip string
      string[] copii = { "Ionel", "Viorel", "Alinel",
                              "Dorel" };
      // Sursa de date este tabloul
      checkedListBox1.DataSource = copii;
}
```

5. Definim clasa, `Elev`, în fişierul `Form1.cs`. Aveţi grijă, *Visual C# Express 2008* cere ca definiţia oricărei clase să urmeze definiţiei formei:

```
class Elev
{
     private string nume;
     public Elev(string n) { nume = n; }
     public string Nume
     {
          get { return nume; }
     }
}
```

Vom defini de asemenea un câmp privat **el** în clasa `Form1`, de tip referinţă la un tablou de obiecte de tip `Elev`:

```
private Elev[] el;
```

6. Tratăm evenimentul **Click** pentru butonul **listButton**. Acţionaţi dublu click pe acesta. Introduceţi acest cod în metoda de tratare:

```
private void listButton_Click(object sender,
                              EventArgs e)
{
    // Membrul clasei Elev care va fi afişat în listă
    // ca ultim item
    listBox1.DisplayMember = "Nume";

    // Tabloul de referinţe de tip object poate reţine
    // referinţele ORICĂROR TIPURI de obiecte
    listBox1.Items.AddRange(new object[]{ "UNU", "DOI",
                        "TREI", new Elev("Nelutu") });
}
```

7. Tratăm evenimentul **Click** pentru butonul **comboButton**. Acţionaţi dublu click pe acesta. Completaţi astfel metoda de tratare:

```
private void comboButton_Click(object sender,
                               EventArgs e)
{
  /* Începând de la apelul BeginUpdate() şi până la
    EndUpdate() controlul nu se mai redesenează, ceea
    ce sporeşte viteza de adăugare în listă */
    comboBox1.BeginUpdate();
    for (int i = 0; i < 20; i++)
       comboBox1.Items.Add("Itemul " + i);
    comboBox1.EndUpdate();
}
```

8. Tratăm evenimentul **Click** pentru butonul **checkListButton1**. Acţionaţi dublu click pe acesta. Completaţi astfel metoda de tratare:

```
private void checkListButton1_Click(object sender,
                                    EventArgs e)
{
      string[] copii = { "Ionel", "Viorel", "Alinel",
                         "Dorel" };
      checkedListBox1.DataSource = copii;
}
```

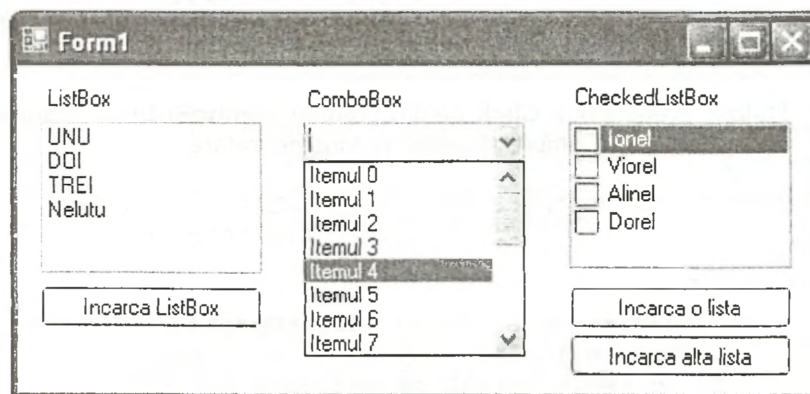
9. Tratăm evenimentul **Click** pentru butonul **checkListButton2**. Acţionaţi dublu click pe acesta. Completaţi în acest mod metoda de tratare:

```
private void checkListButton2_Click(object sender,
                                    EventArgs e)
{
     // Creăm obiectele care se inserează în listă
     el = new Elev[] { new Elev("marius"),
                       new Elev("lenuta")};
```

```
        // Sursa de date este el
        checkedListBox1.DataSource = el;

        // Membrul clasei Elev care se va afişa
        // în listă este proprietatea Nume
        checkedListBox1.DisplayMember = "Nume";
}
```

10. Compilaţi şi rulaţi cu **F5**.



Cele trei controale se încarcă *runtime*, prin acţionarea butoanelor corespunzătoare.

Controalele TrackBar, NumericUpDown şi DomainUpDown

Aceste controale se numesc şi controale de domeniu, deoarece utilizatorul poate introduce un număr limitat de valori dintr-un domeniu prestabilit. Introducerea datelor se face în mod vizual, acţionând de regulă cu mouse-ul.

TrackBar



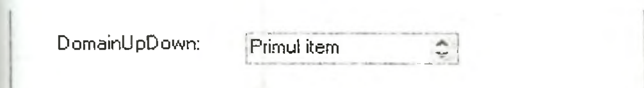
Este un control *scroll*-abil, cu ajutorul cărui utilizatorul poate alege o valoare a proprietăţii `Value`, cuprinsă între o valoare maximă şi minimă. Limitele extreme se impun prin atribuirea de valori proprietăţilor **Minimum** şi **Maximum**. Implicit, aceste valori sunt **0** şi **10**.

NumericUpDown



Este un control care conține o singură valoare numerică, pe care utilizatorul o poate modifica acționând click pe butoanele *Up* sau *Down* ale controlului. Utilizatorul poate să şi introducă valori, dacă proprietatea `ReadOnly` este setată `true`.

DomainUpDown



Este similar cu un **ListBox** în aceea că oferă o listă de opțiuni între care utilizatorul poate alege. Diferența față de **ListBox** este că un singur item este vizibil la un moment dat, iar utilizatorul poate naviga în listă doar acționând săgețile *Up* şi *Down* ale controlului. Pentru crearea colecțiilor de obiecte care vor alcătui lista, se utilizează metodele `Add()`, la fel ca pentru celelalte controale de tip listă.

Transparența şi opacitatea controalelor

`Opacity` şi `Transparency` sunt două proprietăți ale formelor. Au rolul de a regla gradul de trasparență a formei, dar şi a controalelor care se găsesc pe suprafața formei. Când proprietatea `Opacity` este setată la valoarea **1,00** (100%), atunci forma este perfect vizibilă. Dacă valoarea este **0**, atunci forma, inclusiv bordurile, marginile şi controalele de pe formă sunt invizibile. Valori intermediare, reglează niveluri proproționale de tranparență.

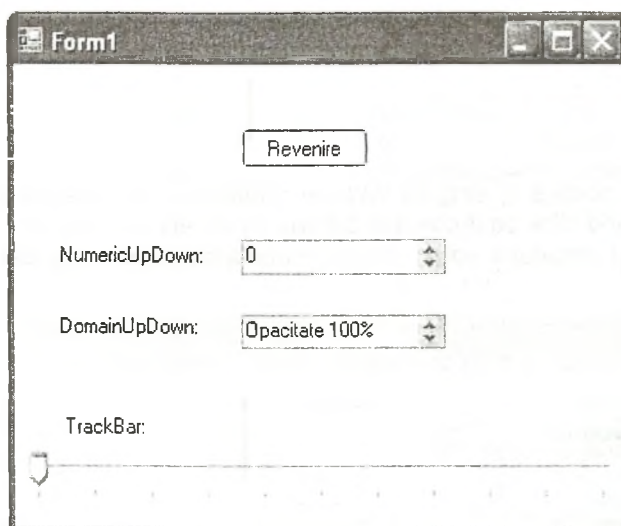
În ce priveşte proprietatea `Transparency`, acesteia îi este ataşată o culoare. Întreaga arie a formei care are aceeaşi culoare (proprietatea `BackColor`) devine trasparentă, iar orice acțiune cu mouse-ul pe suprafața devenită transparentă, este transferată ferestrei de sub aria devenită trasparentă. Această caracteristică vă permite să definiți controale cu forme neregulate.

Aplicația *Opacity*

Vom utiliza controalele `TrackBar`, `NumericUpDown` şi **DomainUpDown** pentru a regla opacitatea unei forme. Evident, unul singur dintre cele trei controale ar fi fost suficient, însă scopul nostru este de a învăța să le utilizăm pe toate.

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *Opacity*.

2. Din **Toolbox** plasați pe formă cu ajutorul mouse-ului un control **NumericUpDown**, un control **DomainUpDown** şi un control **TrackBar**. Adăugați de asemenea trei **Label**-uri pe care le veți eticheta corespunzător controalelor şi un buton căruia setați-i proprietatea **Text**: la *Revenire*:



3. Selectați controlul **DomainUpDown**. Din fereastra **Properties**, atribuiți proprietății **Text** valoarea *"Opacitate 100%"*. Selectați proprietatea **Items**, apoi apăsați butonul din dreapta. În dialogul *String Collection Editor*, introduceți **10** itemi, câte unul pe linie, cu textul: *"Opacitate 100%"*, *"Opacitate 90%"*, ... *"Opacitate 0%"*.

4. Tratăm evenimentul **Scroll** pentru **TrackBar**. Dublu click pe control. Scrieți codul marcat în **Bold**, în *handler*-ul de eveniment:

```
private void trackBar1_Scroll(object sender,
                              EventArgs e)
{
     // Odată cu creşterea valorii Value de la 0
     // 10, opacitatea formei trebuie să scadă
     this.Opacity = 1 - (double)trackBar1.Value / 10;
}
```

5. Tratăm evenimentul **SelectedIndexChanged** pentru controlul **DomainUpDown**, în scopul de a modifica opacitatea în raport cu indexul itemului. Dublu click pe control. Scrieți codul:

```
private void domain_SelectedItemChanged(object sender,
                                        EventArgs e)
{
    // Opacitatea formei scade cu creşterea
    // indexului itemului selectet
    this.Opacity = 1 -
            (double)domainUpDown1.SelectedIndex / 10;
}
```

6. Tratatăm evenimentul **ValueChanged** pentru controlul **NumericUpDown**, pentru a modifica opacitatea în funcție de valoarea afişată în control. Acționați dublu click pe control. Scrieți codul:

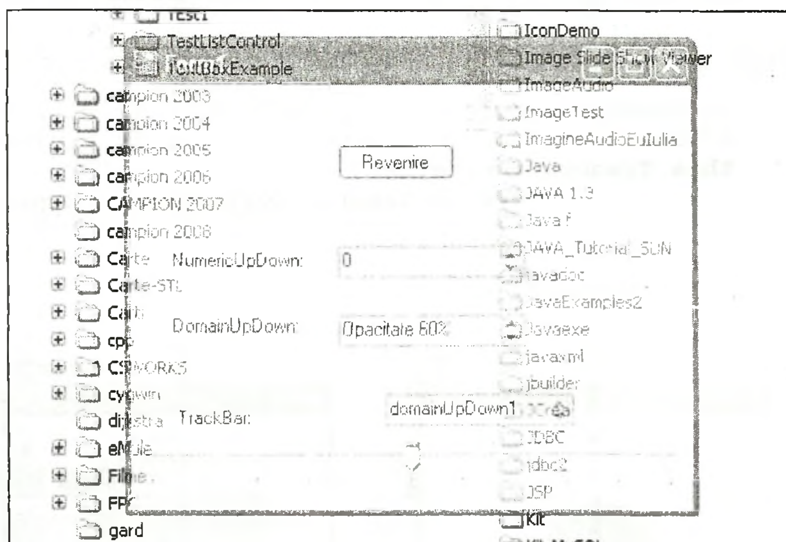
```
private void numericUpDown1_ValueChanged(object sender,
                                         EventArgs e)
{
    // Value este valoarea afişată în control
    this.Opacity = 1 - (double)numericUpDown1.Value/10;
}
```

7. Pentru reiniţializarea valorilor controalelor, dar şi pentru readucerea formei la opacitate maximă, tratăm evenimentul **Click** pentru butonul *"Revenire"*. Dublu click pe buton. Scrieţi codul marcat în **Bold**, în *handler*-ul de eveniment:

```
private void button1_Click(object sender, EventArgs e)
{
     this.Opacity = 1;       // Forma devine opacă

     // Reiniţializarea valorilor în controale
     trackBar1.Value = 0;
     numericUpDown1.Value = 0;
     domain.Text = "Opacitate 100%";
}
```

8. Compilaţi aplicaţia şi rulaţi cu **F5**.



Opacitatea formei se poate seta în mod independent cu ajutotorul celor trei controale.

Aplicaţia *Transparency*

Atunci când doriţi ca anumite porţiuni ale formei să fie transparente şi acţiunea mouse-ului pe suprafaţa transparentă să se transfere ferestrelor de sub formă, utilizaţi proprietatea **Transparency** a formelor.

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *Opacity*.

2. Din **Toolbox** plasați pe formă cu ajutorul mouse-ului un control de tip **Panel** şi două butoane.

3. Selectați panoul şi în fereastra **Properties** atribuiți proprietății **BackColor** valoarea *Gray*. Setați pentru cele două butoane proprietatea **Text** la valorile: "*Panou Transparent*", respectiv "*Forma transparenta*".

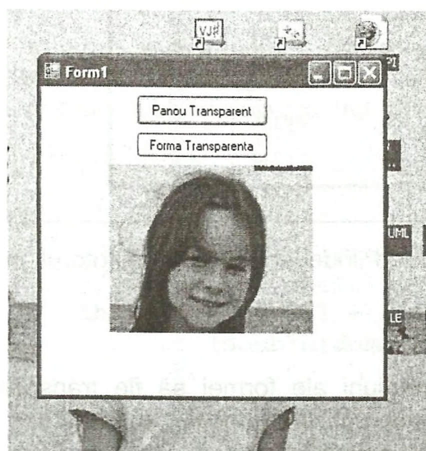
4. Tratăm evenimentul **Click** generat de butonul cu eticheta "*Panou Transparent*". Faceți dublu click pe buton. Scrieți în metoda de tratare următorul cod :

```
private void button1_Click(object sender, EventArgs e)
{
    // Toate zonele formei de culoare gri,
    // devin transparente (suprafaţa panoului)
    this.TransparencyKey = System.Drawing.Color.Gray;
}
```

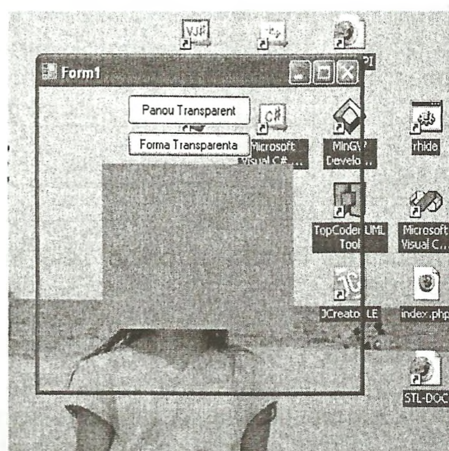
5. Tratăm evenimentul **Click** generat de butonul cu eticheta "*Forma Transparenta*". Dublu click pe buton, iar în metoda de tratare scrieți:

```
private void button2_Click(object sender, EventArgs e)
{
    // Zonele de culoarea implicită a controalelor
    // devin transparente(exteriorul controlului Panel)
    this.TransparencyKey =
                System.Drawing.SystemColors.Control;
}
```

6. Compilați şi executați cu **F5**.



Panoul e transparent



Forma e transparentă

Observaţi că numai suprafaţa formei devine transparentă, nu şi marginile, bara de titlu, sau celelalte controale de pe formă.

Controlul ProgressBar

Controlul nu permite nici o acţiune a utilizatorului. Este destinat să indice în mod vizual progresul unei aplicaţii care decurge lent, aşa cum sunt printarea unor documente, copierea unui număr mare de fişiere, scanarea unui disc.

Stilul de afişare al controlului este determinat de proprietatea **Style**. Există trei stiluri:

- O bară continuă care se umple de la stânga la dreapta
- Blocuri segmentate care progresează de la stânga la dreapta.
- Blocuri care oscilează.

Proprietatea **Value** este valoarea care reprezintă progresul curent al aplicaţiei. Poziţia curentă a barei de progres este în legătură directă cu **Value**. Proprietăţile **Minimum** şi **Maximum** sunt limitele de valori pentru **Value**. **Minimum** este de regulă setat la 0, iar **Maximum** este setat la o valoare care indică terminarea operaţiei.

Întreaga operaţie pentru care vizualizaţi progresul este formată dintr-un număr de operaţii pe care le consideraţi elementare. După fiecare operaţie elementară, cu ajutorul metodei **PerformStep()** incrementaţi valoarea barei de progres cu valoarea **Step**.

Exemplu de utilizare:

```
ProgressBar p = new ProgressBar();
// Maximum poate fi de exemplu numărul de fişiere
// care trebuie copiate
p.Maximum = 500;
p.Mimnimum = 500;
p.Value = 0;
p.Step = 5;

for (int i = p.Minimum; i < p.Maximum; i += p.Step)
{
    // Operaţie elementară (de exemplu, copierea unui fişier)
    // Incrementează bara de progres
    progress.PerformStep();
}
```

Aplicaţia pe care o vom prezenta în cadrul temei ***Controlul Timer*** ilustrează şi utilizarea unui *progress bar*.

Controlul Timer

Contrulul funcționează ca şi cronometru. Declanşează un eveniment la un interval de timp specificat de către programator. **.NET** implementează câteva cronometre dintre care discutăm două:

1. Primul, este definit de clasa **Systems.Windows.Forms.Timer**. Se utilizează în aplicații de tip *Windows Forms*. El apare în mod implicit în **Toolbox** şi trebuie afişat într-o fereastră. Se utilizează numai în medii cu un singur *thread*. Acuratețea este de aproximativ **55** milisecunde.
2. Al doilea, definit de clasa **System.Timers.Timer**, este mai performant. Poate fi folosit în medii *multithread*. Nu se găseşte în mod implicit în **Toolbox**, însă poate fi adus acolo, aşa cum vom vedea.

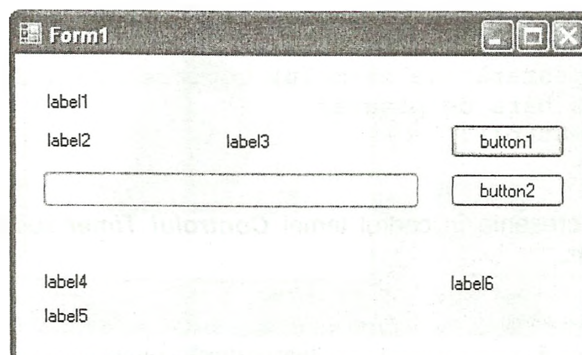
Există asemănări şi deosebiri între cele două *timer*-e. Ambele declanşează un eveniment la un interval specificat. Primul declanşează evenimentul **Tick**, iar al doilea, evenimentul **Elapsed**. Ambele au metodele **`Start()`** şi **`Stop()`**, şi proprietatea **`Enabled`**.

Aplicația *Timer.Example*

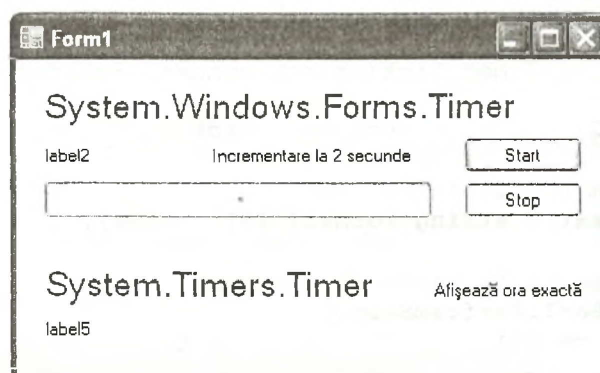
Aplicația execută sarcinile:

- Utilizează *timer*-ul **System.Windows.Timer** care declanşează un eveniment **Tick** la interval de două secunde, timp de 20 de secunde. Numărul de secunde se afişează într-un control **Label**, iar progresul se vizualizează cu o bară de progres. Cronometrul poate fi pornit şi oprit cu butoanele *Stop* şi *Start*.
- Utilizează *timer*-ul **System.Timers.Timer** pentru afişarea orei, într-un control **Label**. Evenimentul **Elapsed** se declanşează la interval de o secundă.

1. Creați un proiect de tip ***Windows Forms Application***, cu numele *TimerExample*.
2. Aduceți din **Toolbox** pe suprafața formei un control de tip **Timer**, două controale de tip Button, un **ProgressBar** şi şi şase controale **Label**:

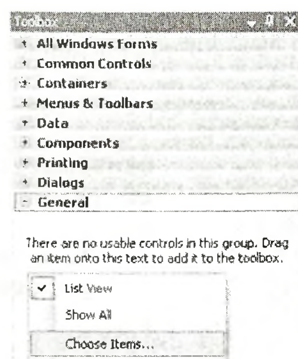


3. Vom seta mai întâi etichetele butoanelor:
   a. Selectaţi `label1`. În fereastra **Properties**, setaţi fontul la dimensiunile (**16**, **50**). Atribuiţi proprietăţii `Text` valoarea "*System.Windows.Forms.Timer.*" Selectaţi `label2`. Setaţi fontul la dimensiunile (**16**, **50**). Ştergeţi numele etichetei. Selectaţi `label3`. Atribuiţi proprietăţii `Text` valoarea "*Incrementare la 2 secunde*".
   b. Selectaţi `label4`. Setaţi fontul la dimensiunile (**16**, **50**). Atribuiţi proprietăţii `Text` valoarea "*System.Timers.Timer*". Selectaţi `label5`. Setaţi fontul la dimensiunile (**16**, **50**). Ştergeţi numele etichetei. Selectaţi `label6`. Atribuiţi proprietăţii `Text` valoarea "*Afişează ora exactă*".
   c. Selectaţi `button1`. Atribuiţi proprietăţii `Text` valoarea "*Start*". Selectaţi `button2`. Atribuiţi proprietăţii `Text` valoarea "*Stop*".



4. Selectaţi referinţa `timer1` în *designer tray*. În panoul **Properties**, setaţi proprietatea **Name** la valoarea `winTimer`. Setaţi la **1000** (o secundă) valoarea proprietăţii `Interval`. Este intervalul la care se declanşează evenimentul **Tick**.

5. Vom adăuga un *timer*-ul din spaţiul de nume **System.Timers**. În **Toolbar**, în partea cea mai de jos, unde nu mai sunt controale, faceţi click drept şi alegeţi *Choose Items…*



6. În dialogul *Choose Toolbox Items*, alegeţi din tab-ul *.NET Framework Components*, clasa **Timer** din spaţiul de nume **System.Timers**. Un icon tip ceas apare în partea de jos în **Toolbar**.

7. Trageţi noul **Timer** pe suprafaţa formei. În *designer tray*, selectaţi referinţa `timer1`. În panoul **Properties**, schimbaţi numele referinţei la `systemTimer`, setaţi

valoarea proprietății **Enabled** la **true**, iar a proprietății **Interval** la **1000**. Aceasta înseamnă că evenimentul **Elapsed** se declanşează la interval de o secundă.

8. Selectați bara de progres. În panoul **Properties**, setați proprietatea **Step** la valoarea **2** şi proprietatea **Maximum** la valoarea **20**.

9. În fişierul *Form1.cs*, în clasa **Form1**, declarați un câmp privat, numit **time**, care contorizează numărul de secunde scurse. În ***Solution Explorer***, faceți click drept pe *Form1.cs* şi selectați *View Code*. Scrieți:

```
private int time = 0;
```

10. Tratăm evenimentul **Tick** pentru **winTimer**. Selectați referința **winTimer** în *designer tray*, apoi în **Properties**, acționați dublu click pe evenimentul **Tick**. Scrieți codul:

```
private void winTimer_Tick(object sender, EventArgs e)
{
    time += 2;     // Un eveniment Tick la 2 secunde

    // Afişăm timpul în etichetă
    label2.Text = string.Format("{0}", time);

    // Incrementarea barei de progres
    progressBar1.PerformStep();
    if (time == 20)       // Dacă s-au scurs 20 secunde
    {
        // oprim timer-ul şi bara de progres
        winTimer.Stop();
        progressBar1.Enabled = false; // oprim
    }
}
```

11. Tratăm evenimentul **Click** pentru butonul *Start*. Faceți dublu click pe buton. Introduceți codul:

```
private void button1_Click(object sender, EventArgs e)
{
    winTimer.Enabled = true;  // Activăm timer-ul
    if (time == 20)           // Dacă au trecut 20 sec
    {                         // se fac reinițializări
        time = 0;
        progressBar1.Value = 0;
    }
}
```

12. Tratăm evenimentul **Click** pentru butonul *Stop*. Faceți dublu click pe buton. Introduceți codul:

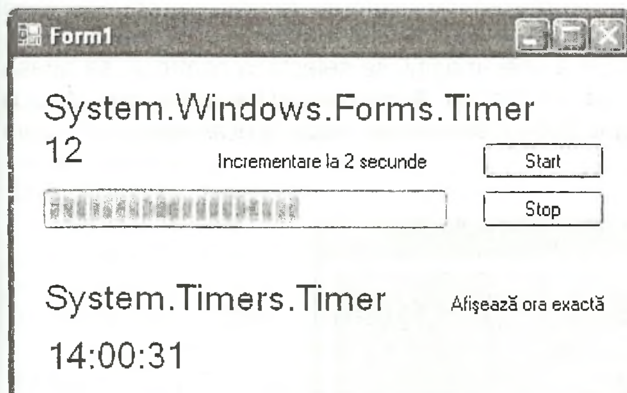
```
private void button2_Click(object sender, EventArgs e)
{
    winTimer.Stop();                    // Oprim cronometrul
    progressBar1.Enabled = false; // Oprim progress bar
    if (nrTick == 20)
    {
        nrTick = 0;
        progressBar1.Value = 0;
    }
}
```

13. Tratăm evenimentul **Elapsed** pentru **systemTimer**. Selectați referința **systemTimer** în *designer tray*, apoi în **Properties**, acționați dublu click pe evenimentul **Elapsed**. Scrieți codul:

```
private void systemTimer_Elapsed(object sender,
                        System.Timers.ElapsedEventArgs e)
{
   // Proprietatea Now a clasei DateTime, returnează
   // ora exactă
   label5.Text =System.DateTime.Now.ToLongTimeString();
}
```

14. Compilați și rulați aplicația cu **F5**.

Captură de ecran la execuție:

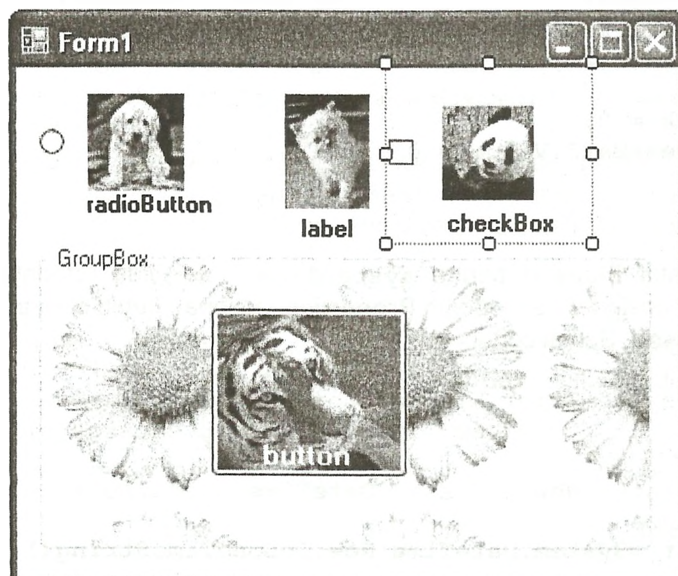


Controalele PictureBox și Imagelist

Imagini de fundal pentru controale

Multe controale suportă afișarea de imagini, deoarece moștenesc proprietatea **BackgroundImage** de la clasa **Control**. Între aceste controale,

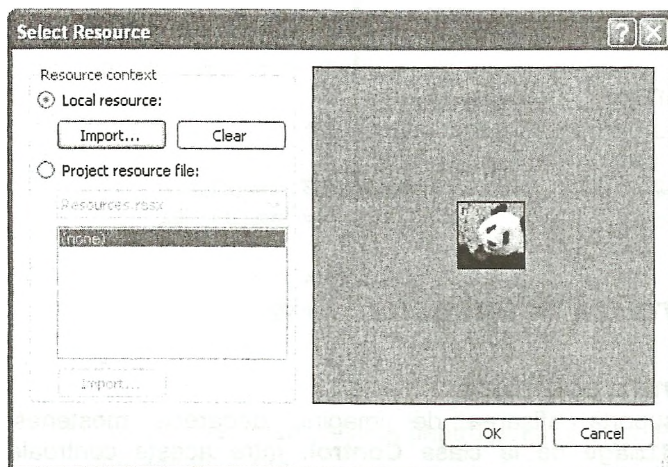
menţionăm **Button**, **CheckBox**, **RadioButton**, **PictureBox**. Şi controalele de tip container cum sunt **Panel**, **GroupBox** şi **Form** suportă imagini de fundal.



Anumite controale suportă şi imagini de prim plan. De exemplu, **Button** are în acest scop proprietatea `Image`, moştenită de la clasa **ButtonBase**. În cazul în care utilizaţi atât o imagine de fundal cât şi de prim plan, imaginea de prim plan se afişeză deasupra.

Controalele au în plus proprietăţi cu ajutorul cărora se pot modifica caracteristicile imaginilor afişate.

Pentru afişarea *design time* a unei imagini, se selectază controlul, se apasă butonul cu etichta "**...**" din linia proprietăţii `BackgroundImageLayout` sau a proprietăţii `Image`. În dialogul *Select Resource*, bifaţi "*Local resource*", apoi apăsaţi butonul "*Import*".



Caracteristicilor imaginilor se modifică simplu din panoul **Properties** atribuind valori următoarelor proprietăți:

- `BackgroundImageLayout`
  Are valorile posibile: `None`, `Tile`, `Center`, `Strech` şi `Zoom`. De exemplu, mai sus, pentru CheckBox, şi RadioButton am ales `Center`, iar pentru **GrupBox** valoarea `Tile`.
- `ImageAllign`
  Label-urile nu suportă `BackgroundImage` deci utilizați `Image` pentru afişare, iar pentru modul de afişare, alegeți dintre valorile disponibile în panoul Properties, pentru proprietatea `ImageAllign`.
- `AutoSize`
  Pentru *label*-uri, *checkbox*-uri şi radiobutoane trebuie să setați această proprietate la valoarea false, altminteri nu veți reuşi să redimensionați suprafețele controalelor, astfel încât să potrivească imaginile.
- `TextAllign`
  Această proprietate vă permite să stabiliți poziția şi alinierea textului în raport cu suprafața controlului.

Clasa Image

.NET defineşte clasa **Image** în spațiul de nume **System.Drawing**, pentru lucrul cu imagini.
Clasa este abstractă, deci nu se pot crea obiecte de acest tip. În schimb, imaginile se citesc din fişiere cu metoda statică `FromFile()`:

Exemplu:

```
Image im = Image.FromFile(
        Path.Combine(Application.StartupPath, "foto.gif"));
Label lbl = new lbl Label();  // Creează o etichetă
lbl.Image = im;       // Afişează imaginea în control
```

Codul de mai sus încarcă imaginea din fişierul `"foto.gif"` aflat în folderul în care se găseşte fişierul *assembly* (exe) al aplicației. Metoda nu recunoaşte decât formatele **.GIF** şi **.BMP**. Eticheta `lbl` afişează imaginea încărcată.

Clasa are proprietăți şi metode pentru manipularea imaginilor. Metoda `FromStream()` încarcă imagini de diferite tipuri nu numai în formatele standard **.GIF** sau **.BMP**. Creează un stream din care puteți descărca imaginea. *Stream*-ul poate să fie de conectat la orice dispozitiv de intrare, de exemplu la o adresă de Internet:

```
string s = http://......./foto.jpg;  // URL
// Sunt necesare obiectele wrq şi wrs pentru crearea
// stream-ului
WebRequest wrq = WebRequest.Create(s);
WebResponse wrs = wrq.GetResponse();
Stream str = wrs.GetResponseStream(); // Crează stream-ul
```

```
Label lbl = new Label();

// Afişează imaginea într-un control de tip Label
lbl.Image = Image.FromStream(str);
str.close();
```

Vom reveni asupra tipului **Image**.

Controlul PictureBox

PictureBox este un control aflat la îndemână în **Toolbox**. E folosit pentru afişarea imaginilor grafice, obţinute din fişiere **JPEG**, **GIF**, **BMP**, **PNG**, iconuri sau metafişiere.

Prezentăm câţiva membri reprezentativi ai clasei `PictureBox`.

Proprietăţi

`Image`	Se setează pentru afişarea *design time* sau *run time* a imaginilor
`SizeMode`	Specifică modul de afişare a imaginii. Are valorile: `StrechImage`, `CenterImage`, `Normal`, `Zoom`.
`ClientSize`	Permite modificarea dimensiunii imaginii în timpul rulării.
`ImageLocation`	Reprezintă calea sau URL-ul spre imaginea care se afişează în **PictureBox**.

Metode

`Load()`	Afişează imaginea specificată de proprietatea `ImageLocation`. Are două versiuni.
`SetBounds()`	Setează limitele controlului, la o anumită locaţie, şi la dimensiunile specificate.

Evenimente

`Click`	Se declanşează la click pe control
`Paint`	Se declanşează când controlul trebuie redesenat
`ClientSizeChanged`	Se declanşează când proprietatea **ClientSize** se modifică

Resursele unui proiect Visual C#

Aţi văzut că un program C# poate accesa fişiere externe de tip imagine. problema apare când asemenea resurse sunt şterse sau mutate din greşeală. În aceste condiţii aplicaţia nu le mai poate utiliza.

În Visual C# există posiblitatea să incorporaţi resurse de care aplicaţia are nevoie: imagini, iconuri, sunete, stringuri în acelaşi fişier *assembly*(.exe) care conţine codul compilat al aplicaţiei. Pentru a vedea cum se realizează acest lucru, realizăm un mic proiect.

Aplicația *ImageResourceExample*

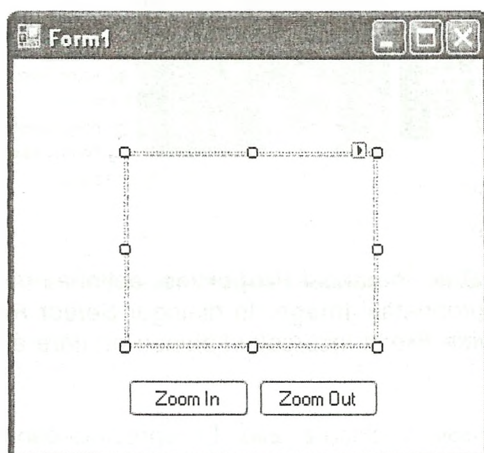
Proiectul va fi unul simplu: pe o formă, avem un control **PictureBox**. Imaginea pe care o afişăm în control, va fi una luată din resursele proiectului. Pentru divertisment, implementăm şi facilitatea de mărire şi de micşorare a imaginii (*Zoom*).

15. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *ImageResourceExample*.

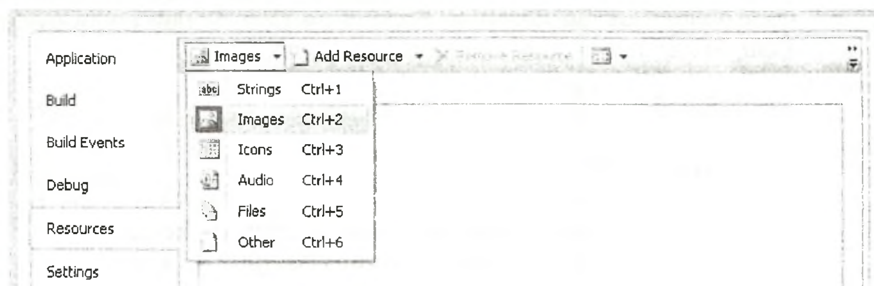
16. Aduceți din **Toolbox** pe suprafața formei un control de tip **PictureBox** şi două butoane.

17. Selectați controlul **PictureBox**. În panoul **Properties**, setați pentru proprietatea **`SizeMode`** valoarea `Zoom`. Aceasta permite controlului să se redimensioneze odată cu imaginea.

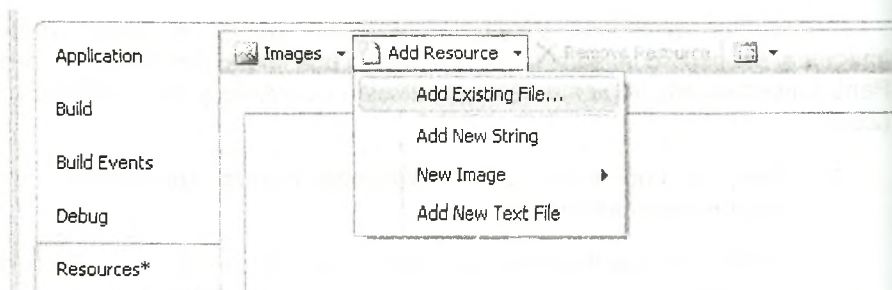
18. Selectați pe rând cele două butoane şi stabiliți pentru proprietatea **Text**, valorile *ZoomIn*, respectiv *ZoomOut*:



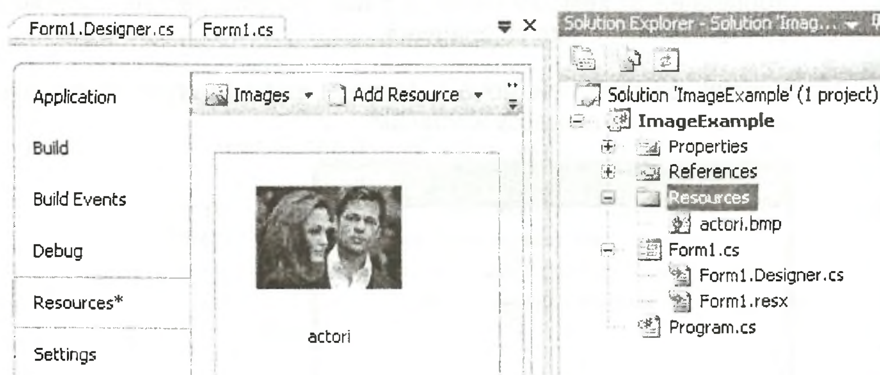
19. Acum adăugăm o imagine ca resursă a proiectului. În ***Solution Explorer***, dublu click pe ***Properties***. Se deschide fereastra ***Editorului de Resurse***. Pe coloana din stânga ferestrei, selectați *Resources*. Din lista primului *drop down button* din stânga sus, selectați *Images*:



20. Din lista butonului ***Add Ressource***, selectați *Add Existing File ...*:



21. Selectați o imagine pe care ați pregătit-o în acest scop:



22. Selectați controlul **PictureBox**. În panoul **Properties**, acționați butonul cu eticheta (...) din dreapta proprietății **Image**. În dialogul ***Select Resource*** veți selecta *Project Resource File* și încărcați imaginea pe care ați adus-o la resursele proiectului.

23. Declarați un câmp privat `zoom`, în clasa `Form1`. El reprezintă cantitatea cu care se incrementează dimensiunile controlului la fiecare apăsare a butoanelor:

```
private int zoom;
```

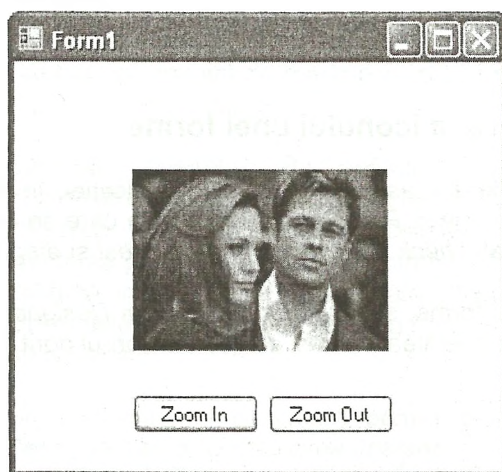
24. Tratăm evenimentul **Click** pentru butonul cu textul *ZoomIn*. Acționați dublu click asupra lui. Se deschide *Editorul de Cod*. Scrieți codul evidențiat cu *Bold*, în handler-ul de eveniment:

```
private void button1_Click(object sender, EventArgs e)
{
      zoom = 2;
      // Lățimea și înălțimea după o apăsare de buton
      int W = pictureBox1.Size.Width + zoom;
      int H = pictureBox1.Size.Height + zoom;
```

```
        // Are loc actualizarea dimeniunilor prin
        // modificarea proprietății ClientSize
        pictureBox1.ClientSize = new Size(W, H);
}
```

25. Tratăm evenimentul **Click** pentru butonul cu textul *ZoomOut*. Acționați dublu click asupra lui. În *handler*-ul evenimentului veți scrie același cod, cu singura diferență că prima linie este: `zoom = -2;` (dimensiunile scad).

26. Compilați și rulați cu **F5**.



Prin apăsarea celor două butoane, se obține efectul de mărire, respectiv de micșorare a imaginii.

De reținut:

- Resursele adăugate proiectului se integrează în mod automat în fișierul *assembly* compilat (fișierul .exe).
- Resursele sunt imagini, iconuri, sunete, sau stringuri.
- Editarea resurselor se face *design time* cu ajutorul *Editorului de Resurse*, sau *runtime* în mod programatic.
- Între folderele proiectului apare unul nou, numit **Resources**, care poate fi vizualizat din ***Solution Explorer***.
- Se pot elimina elemente folderul *Resources*, fără ca aceasta să afecteze fișierul *assembly*. La fiecare ***build***, compilatorul consultă acest folder și numai dacă a apărut o versiune nouă a unui fișier, acesta este integrat în *assembly*.

Manipularea *runtime* a resurselor

Resursele se accesează uşor din cod, deoarece mediul integrat crează o clasă **Resources**, aflată în spaţiul de nume **Properties**, iar acesta din urmă se găseşte în interiorul spaţiului de nume al aplicaţiei. În clasa **Resources**, toate resursele se accesează simplu, ca oricare membru public.

Iată de exemplu cum puteţi atribui ***runtime*** unui control o altă imagine din resursele proiectului:

```
pictureBox1.Image = Image.Properties.Resources.pisica;
```

Resursa pisica este un tip nou creat, pe baza unei imagini obţinute dintr-un fişier care se numea de exemplu *pisica.bmp*.

Setarea iconului aplicaţiei şi a iconului unei forme

Ca să ataşaţi un icon fişierului ***assembly*** (.exe) al aplicaţiei, în ***Solution Explorer*** acţionaţi dublu click pe itemul *Properties*. În fereastra care se deschide, selectaţi *Application*, apoi selectaţi *check box*-ul *Icon and manifest* şi alegeţi iconul dorit.

Ca să ataşaţi un icon unei forme, selectaţi forma în *Form Designer*, apoi în fereastra **Properties** selectaţi proprietatea **Icon** şi alegeţi un icon-ul dorit.

Clasa ImageList

ImageList este un control care încapsulează o colecţie de imagini sau icon-uri. Imaginile se folosesc de către alte controale, cum este **ListView**, **TreeView**, sau **ToolStrip**. Principalii membrii ai clasei sunt:

Proprietăţi

`Images`	Colecţia de imagini utilizate de către alte controale
`ImageSize`	Dimensiunea imaginilor în colecţie. Indiferent de dimensiunile iniţiale ale imaginilor, acestea se convertesc la un fomat specificat în momentul în care se adaugă la colecţie
`ColorDepth`	Valoare care indică adâncimea de culoare. Valorile obişnuite sunt 5 biţi (256 culori), 16 biţi (*high color*), 24 biţi (*true color*).

Metode

`Draw()`	Metodă supraîncărcată care desenează imaginile pe suprafaţa unui control.

Aplicația ImageListExample

Vom utiliza un control de tip **ImageList** ca sursă de imagini pentru un control **PictureBox**. Rețineți că un **PictureBox** nu se poate conecta direct la întreaga listă de imagini, aşa cum o fac alte controale mai evoluate, ca **ListView** sau **TreeView**.

În această aplicație, la fiecare acționare unui buton, se accesează câte o imagine din **ImageList** şi imaginea se afişează în **PictureBox**.

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *ImageListExample*.

2. Aduceți din **Toolbox** pe suprafața formei un control de tip **PictureBox** şi un buton.

3. Căutați opt imagini de mărime medie şi salvați-le într-un folder oarecare.

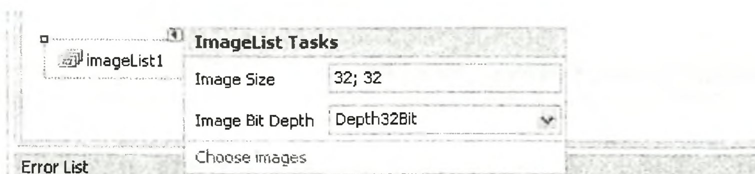
4. Selectați controlul **PictureBox**. În panoul **Properties**, setați pentru proprietatea `SizeMode` valoarea `StrechImage`. Aceasta "întinde" imaginea pe toată suprafața controlului. Setați ca imagine inițială pentru proprietatea `Image`, una dintre cele 8 imagini pe care le-ați pregătit pentru acest proiect:



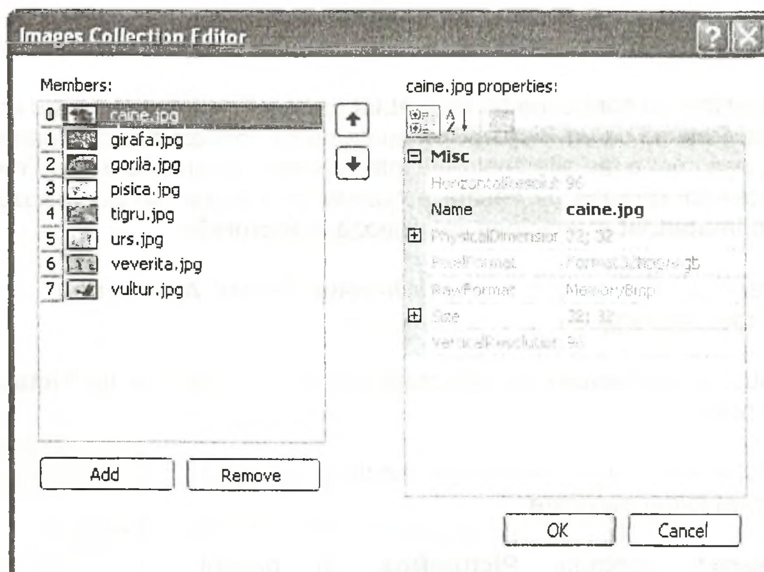
5. Selectați butonul şi stabiliți pentru proprietatea `Text`, valoarea "*Urmatoarea Imagine*".

6. Din **Toolbox**, trageți pe suprafața formei un control de tip **ImageList**. În mod automat, în ***tray***-ul *Form Designer*-ului apare numele referinței: `imageList1`.

7. Apăsați click pe săgeata mică din colțul dreapta sus al numelui `imageList1` în *designer tray*. Stabiliți `ImageSize` la valorile **32** şi **32** şi adâncimea de culoare la valoarea `Depth32Bit`.



8. Alegeți *Choose images*. În *Editorul Colecțiilor de Imagini*, adăugați pe rând câte o imagine la ImageList. Cifrele din stânga reprezintă pozițiile imaginilor în colecție. Pozițiile se pot schimba, acționând săgețile:



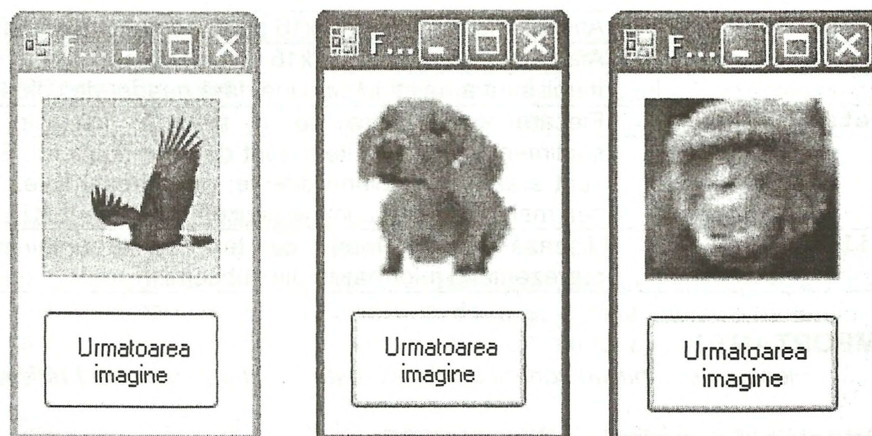
9. Declarați în clasa *Form1*, în fişierul *Form1.cs*, câmpul privat `count`. Acesta reprezintă indexul imaginii din `ImageList`, care se afişează la un moment dat în **PictureBox**:

```
private int count = 0;
```

10. Vom trata evenimentul **Click** pentru buton. Acționați dublu click pe suprafața lui. Scrieți codul evidențiat cu **Bold**, în *handler*-ul de eveniment:

```
private void button1_Click(object sender, EventArgs e)
{
      // Atribuim proprietatății Image imaginea din
      // ImageList corespunzătoare indexului count % 8
      pictureBox1.Image = imageList1.Images[count % 8];
      count++;
}
```

11. Compilați şi lansați în execuție cu **F5**.



Imaginea se schimbă la fiecare click.

De reținut:

- Obiectele de tip **ImageList** rețin colecții de imagini, care se utilizează de către alte controale.
- Toate imaginile se formatează în mod automat la aceleaşi dimensiuni şi adâncimi de culoare precizate în momentul adăugării în **ImageList**, indiferent de formatul inițial.
- Imaginile se accesează cu ajutorul operatorului de indexare. De exemplu, `imageList1.Images[3];` reprezintă a patra imagine din colecția referită de `imageList1`. Atenție la acest exemplu! În colecția aceasta trebuie să existe cel puțin **4** imagini, altfel `Images[3]` duce la aruncarea unei excepții.

Controlul ListView

ListView este un control complex, destinat afişării unei liste de itemi.

Panoul din dreapta al aplicației *Windows Explorer* implementează un **ListView**. *Control Panel* este un alt exemplu cunoscut de către toată lumea.

Pentru fiecare item din listă controlul afişează un text şi un icon.

Itemii se pot afişa în cinci moduri diferite. Cele cinci moduri sunt determinate de valorile pe care le poate avea proprietatea `View`, a clasei **ListView**.

Modurile de afişare într-un ListView

Valorile posibile ale proprietății `View` determină cele cinci moduri de afişare. Aceste valori sunt:

`LargeIcon`	- Afişează iconuri mari (32x32 pixeli), cu text dedesupt, aliniate de la stânga la dreapta şi de sus în jos

`SmallIcon`	- Afişează iconuri mici (16x16 pixeli), cu eticheta în dreapta
`List`	- Afişează iconuri mici (16x16 pixeli), cu eticheta în dreapta. Itemii sunt aranjaţi în coloane, fără *header* de coloană.
`Details`	- Fiecare item apare pe o singură linie. Informaţiile suplimentare despre item sunt date de subitemi. Subitemii sunt aranjaţi pe colane diferite, pe aceeaşi linie. Coloana cea mai din stânga conţine un icon mic şi un text.
`Tile`	- Afişează iconuri mari, cu text în dreapta acestuia, reprezentând informaţiile din subitemi.

IMPORTANT !

*Fiecare item într-un control **ListView**, este o instanţă a clasei **ListViewItem**.*

Principalii membri ai clasei ListView

Clasa conţine numeroase metode, proprietăţi şi evenimente, graţie cărora are multiple calităţi. Iată câteva :

- ✓ Suportă selecţia simplă şi multiplă a itemilor.
- ✓ Tratatând corespunzător a evenimentele de la tastatură şi mouse, itemii pot deveni activi, declanşând acţiuni ca deschidere de fişiere, lansare de aplicaţii, etc.
- ✓ Se poate folosi pentru afişarea informaţiilor dintr-o bază de date, a informaţiilor din fişiere, de pe disc, sau a informaţiilor din aplicaţii.

Principalii membri ai clasei sunt :

Proprietăţi :

`Items`	- Depozitează colecţia de obiecte de tip `ListViewItem` care reprezintă subitemii.
`Collumns`	- Returnează colecţia de obiecte de tip `ColumnHeader` care sunt afişate în **ListViewControl**.
`View`	- Permite specificarea modului de afişare a itemilor în listă
`LargeImageList`	- Indică obiectul `ImageList` care conţine iconurile folosite la afişarea itemilor, atunci când proprietatea `View` are valoarea `LargeIcon`
`SmallImageList`	- Indică obiectul `ImageList` care conţine iconurile folosite atunci când proprietatea `View` are oricare altă valoare în afară de `LargeIcon`
`MultiSelect`	- Când este setat `true`, se pot selecta mai mulţi itemi odată

Metode:

`Clear()`	- Înlătură toţi itemii din control
`BeginUpdate()` `EndUpdate()`	- Prima metodă suspendă redesenarea controlului `ListView` la fiecare adăugare a unui item în situaţia în care se adaugă un mare număr de itemi. - Se apelează a doua metodă după terminarea operaţiei de update, pentru a permite controlului să se redeseneze.

Evenimente:

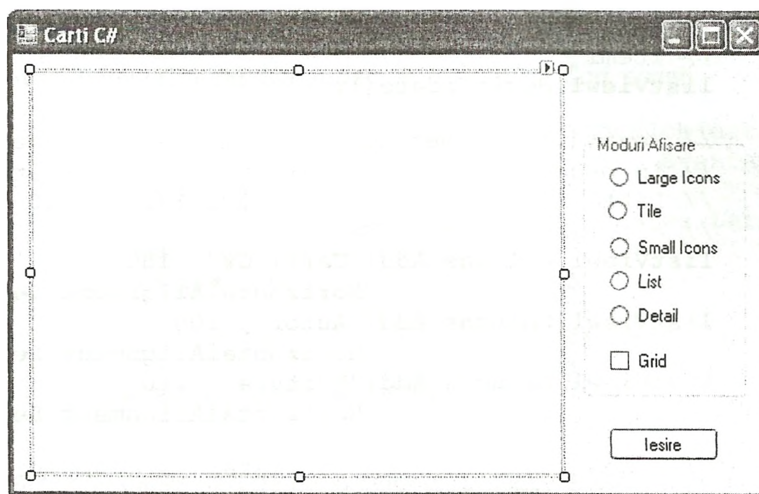
`ColumnClick`	- Se declanşează la click pe header-ul unei coloane
`SelectedIndexChanged`	- Se declanşează când utilizatorul selectează sau deselectează un item

Aplicaţia *ListViewExample*

Proiectul afişează o listă titluri de cărţi, în toate modurile de afişare: *LargeIcon*, *SmallIcon*, *List*, *Tile*, *Detail*. În modul *Detail*, fiecare item (carte) are câte doi itemi suplimentari, afişaţi în coloana a doua şi a treia: numele autorului şi editura care a publicat cartea. De asemenea, tot în modul *Detail* este disponibilă opţiunea de *Grid*.

1. Creaţi un nou proiect de tip ***Windows Forms Application***, cu numele *ListViewExample*.

2. Trageţi din **Toolbox** pe suprafaţa formei un control de tip **ListView** un control **GroupBox**, cinci controale **RadioButton**, un **CheckBox** şi un buton de apăsare şi aranjaţi-le astfel:



3. *Căutaţi câteva fişiere de tip imagine cu coperţi de cărţi şi salvaţi-le într-un folder oarecare. Notaţi-vă pentru fiecare carte, titlul, autorul şi editura.*

4. Aduceţi din **Toolbox** două controlale de tip **ImageList**. În *tray*-ul *Form Designer*-ului apar **`imageList1`** şi **`imageList2`**. Ele reprezintă referinţele spre colecţia de icon-uri mici necesare în modurile **`SmallIcon`**, **`List`** şi **`Detail`**, respectiv spre colecţia de icon-uri mari, necesare în modurile **`LargeIcon`** şi **`Tile`**.

5. Setaţi pentru **`imageList1`**, dimensiunile (**16**, **16**) şi adâncimea de culoare **8** biţi. Setaţi pentru **`imageList2`** dimensiunile (**48**, **48**) (puteţi decide şi alte dimensiuni) şi adâncimea de culoare **32** de biţi:



6. Adăugați aceleaşi imagini în ambele obiecte de tip **ImageList**, acționând *Choose images*.

7. Dorim să populăm controlul **ListView** la încărcarea formei. Faceți dublu click pe suprafața formei, pentru a trata evenimentul **Load**. În corpul *handler*-ului evenimentului, scrieți codul evidențiat în **Bold**:

```
private void Form1_Load(object sender, EventArgs e)
{
      // Controlul listView1 preia colecţiile de iconuri
      // mici şi de iconuri mari
      listView1.SmallImageList = imageList1;
      listView1.LargeImageList = imageList2;

      // Suspendăm redesenarea listei cât timp adăugăm
      // itemi
      listView1.BeginUpdate();

      // Pentru modul Detail, adăugăm trei coloane având
      // în header-e textele "Carti", Autor, "Editura".
      // Lăţimile coloanelor sunt 150, 100 şi 110 şi
      // alinierea orizontală
      listView1.Columns.Add("Carti C#", 150,
                            HorizontalAlignment.Left);
      listView1.Columns.Add("Autor", 100,
                            HorizontalAlignment.Left);
      listView1.Columns.Add("Editura", 110,
                            HorizontalAlignment.Left);

      // Adăugăm primul item în listă. Este un obiect de
      // tip ListViewItem
      ListViewItem it1 = new ListViewItem();

      // Indexul icon-ului din listImage1 şi listImage2.
      // Puteţi avea mai mulţi itemi cu acelaşi index,
      // deci cu acelaşi icon
      it1.ImageIndex = 0;

      // Textul itemului
      it1.Text = "C# 2008 Code Book";

      // Creăm cei doi subitemi. Puteţi avea oricâţi
      // subitemi pentru un item, dar aveţi grijă să
      // creaţi tot atâtea coloane.
```

```
        it1.SubItems.Add("Jürgen Bayer");
        it1.SubItems.Add("Addison-Wesley 2007");

        // Adaugăm itemul în controlul ListView
        listView1.Items.Add(it1);

        // Se crează al doilea item, în acelaşi mod
        ListViewItem it2 = new ListViewItem();
        it2.ImageIndex = 1;
        it2.Text = "Beginning C# 2008";
        it2.SubItems.Add("Karl Watson");
        it2.SubItems.Add("WROX 2008");
        listView1.Items.Add(it2);

        // Creaţi ceilalţi itemi ca mai sus
        // ...

        // Am terminat adăugarea de itemi.
        // Acum permitem controlui să se redeseneze
        listView1.EndUpdate();
}
```

8. Acționați dublu click pe radio butonul cu eticheta *Large Icons*. Tratăm evenimentul **CheckedChanged**. În *Editorul de Cod*, scrieți:

```
private void radioButton1_CheckedChanged(object sender,
                                          EventArgs e)
{
        // Proprietatea View primeşte valoarea LargeIcon.
        // LargeIcon este un membru al enumerării View
        listView1.View = View.LargeIcon;

        // Dezactivăm căsuţa de validare pentru Grid.
        // Efectul Grid e posibil numai în modul Detail
        checkBox1.Enabled = false;
}
```

9. Acționați dublu click pe radio butonul cu eticheta *Tile*. În corpul *handler*-ului de eveniment, scrieți codul:

```
listView1.View = View.Tile;
checkBox1.Enabled = false;
```

10. Acționați dublu click pe radio butonul cu eticheta *Small Icons*. În corpul *handler*-ului de eveniment, scrieți codul:

```
listView1.View = View.SmallIcon;
checkBox1.Enabled = false;
```

11. Acționați dublu click pe radio butonul cu eticheta *List*. În corpul *handler*-ului de eveniment, scrieți:

```
listView1.View = View.Tile;
checkBox1.Enabled = false;
```

12. Acționați dublu click pe checkbox-ul cu eticheta *Grid*. Tratăm evenimentul **CheckedChanged**. În *handler-ul de eveniment*, scrieți codul marcat cu **Bold**:

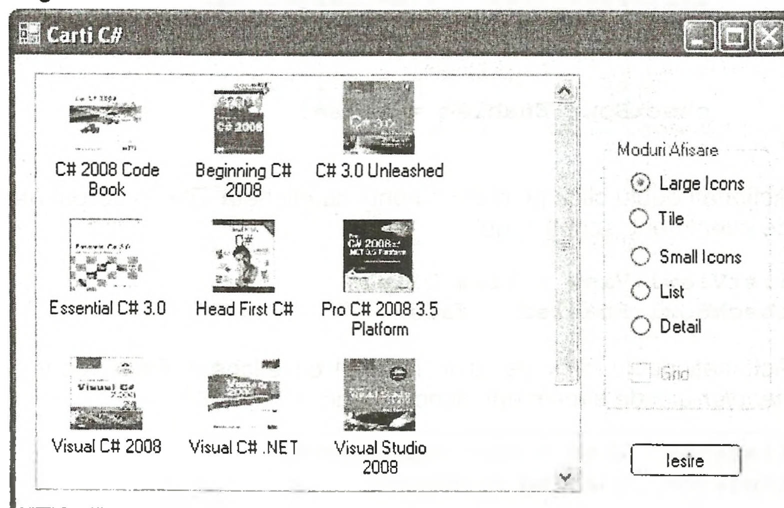
```
private void checkBox1_CheckedChanged(object sender,
                                      EventArgs e)
{
    // Dacă tocmai s-a selectat checkbox-ul
    if (checkBox1.Checked)
    {   // atunci punem griduri.
        listView1.GridLines = true;
    }
    else
    {     // Dacă tocmai s-a deselectat controlul
          // scoatem gridurile
          listView1.GridLines = false;
    }
}
```

13. Acționați dublu click pe butonul *Iesire*. Tratăm evenimentul **Click**. În corpul metodei de tratare, scrieți:

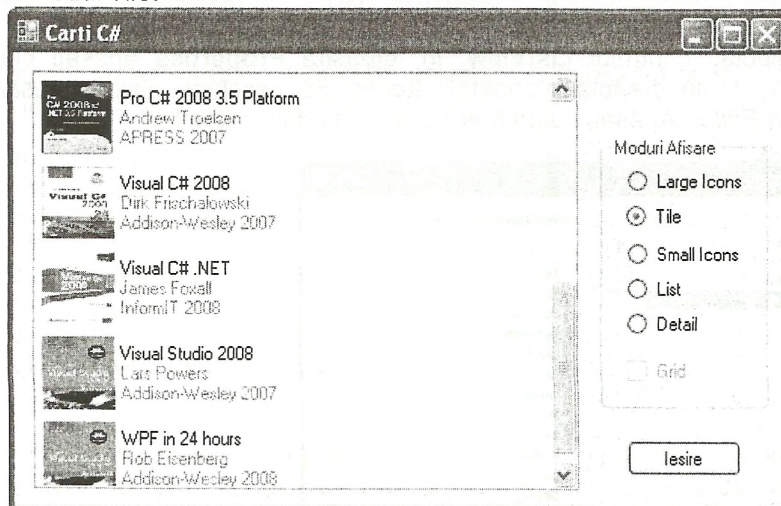
```
Application.Exit();  // Ieşire din aplicaţie
```

14. Compilați proiectul și rulați cu **F5**.

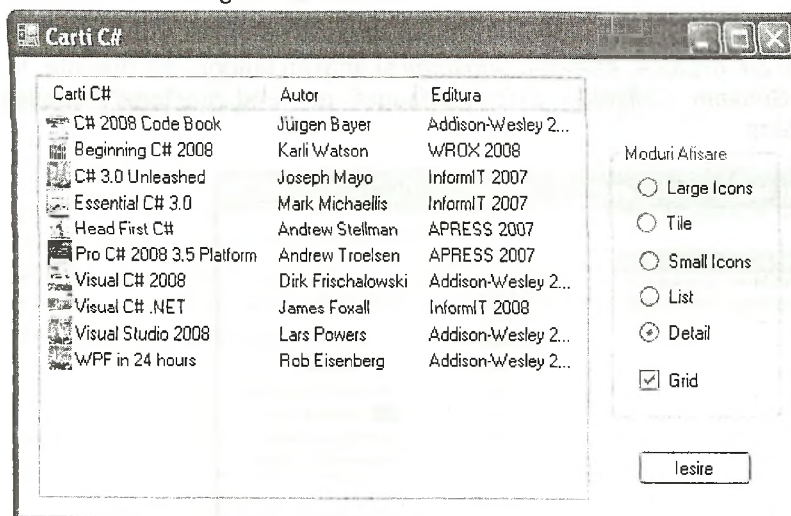
La rulare, se obține:
Vedere *Large Icons*:



Vedere în modul *Tile*:



Vedere în modul *Detail* cu gridurile activate:



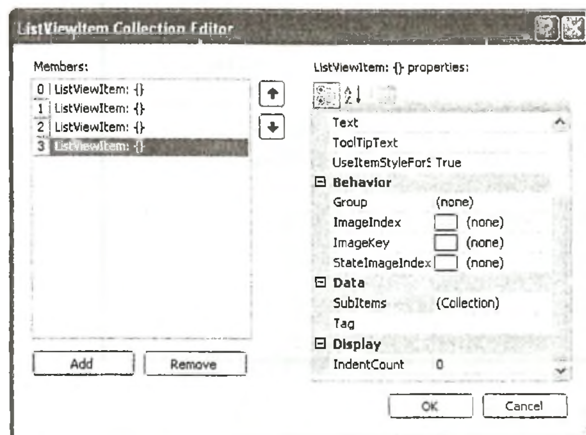
Celelalte moduri de afişare funcţionează de asemenea în mod corespunzător.

Observaţii:

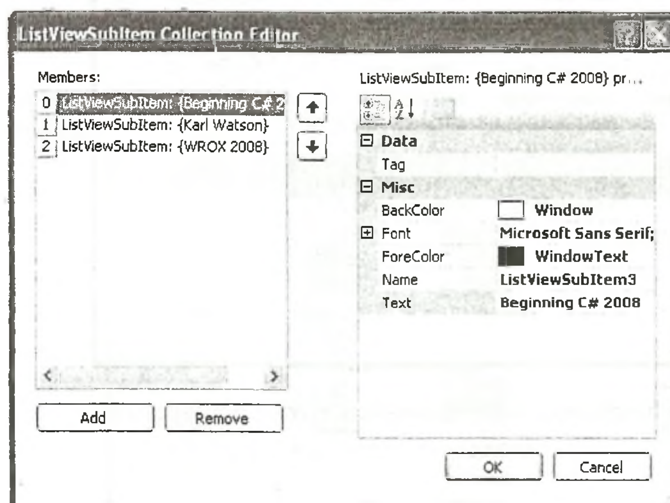
- În multe situaţii itemii nu se introduc în mod manual. Aceştia pot fi încărcaţi din fişiere **XML**, sau din baze de date, aşa cum vom vedea în capitolele următoare.
- În proiectul acesta, itemii au fost adăugaţi programatic. Este una dintre cele mai simple variante, dar nu unica. Mai există şi alte variante de adăugare programatică a itemilor, pe care le puteţi găsi în literatura de specialitate.

Adăugarea *design time* a itemilor într-un ListView

Selectaţi controlul **ListView**. În fereastra **Properties** apăsaţi butonul cu eticheta (...) din dreapta proprietăţii **Items**. Se deschide dialogul *ListViewItem Collection Editor*. Apăsaţi butonul *Add* pentru a adăuga itemi.



În panoul din dreapta, selectaţi *Subitems* şi apăsaţi butonul din dreapta. În dialogul *ListViewSubitem Collection Editor*, adăugaţi pe rând subitemii necesari pentru fiecare item:



De reţinut:

- Dacă implementaţi afişarea *Detail*, atunci este nevoie să adăugaţi coloane (obiecte de tip `ColumHeader`).
- Itemii sunt obiecte diferite de tip `ListViewItem`.
- Numărul subitemilor trebuie să coincidă cu numărul coloanelor.

Probleme propuse

1. Aduceți următoarea facilitate aplicației *ListViewExample*: la apăsarea unui buton, se activează opțiunea de selectare a întregului rând, atunci când utilizatorul face click pe un item în modul *Detail*.

2. În aplicația *ListViewExample*, introduceți câte un mic icon care precedă textul în *header*-ul fiecărei coloane (modul *Detail*).
Indicație: Utilizați constructorul adecvat al clasei `ColumnHeader`, astfel:

```
ColumnHeader c = new ColumnHeader(3);
// 3 este indexul iconului
c.Text = "C# 2008"
listView1.Columns.Add(c);
// etc
```

3. * Creați o aplicație care preia într-un **ListView** informațiile despre fişierele şi folderele care se găsesc într-un folder dat. Folderul se alege cu ajutorul controlului predefinit **FolderBrowserDialog**.

4. Creați o aplicație care permite editarea etichetelor itemilor unui **ListView**.

Controlul TreeView

Controlul afişează o colecție ierarhică de elemente care se numesc noduri. Fiecare nod este o instanță a clasei **TreeNode**. Proprietatea `Nodes` a controlului memorează această colecție de noduri.

Nodurile se crează programatic relativ simplu. Creați un proiect de tip *Windows Forms* şi tratați evenimentul **Load** generat de formă (dublu click pe suprafața ei), apoi scrieți codul:

```
private void Form1_Load(object sender, EventArgs e)
{
      // Instanțiem un TreeView
      TreeView tw = new TreeView();

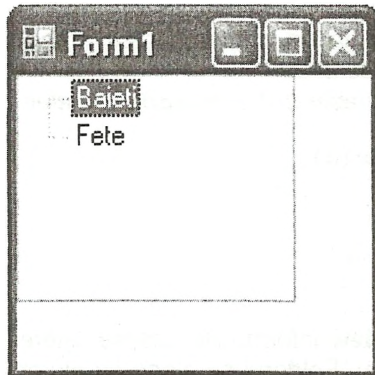
      // Creăm un prim nod cu eticheta "Baieti"
      TreeNode tn1 = new TreeNode();
      tn1.Text = "Baieti";

      // Adăugăm nodul în control
      tw.Nodes.Add(tn1);

      // Creăm al doilea nod cu eticheta "Fete"
      TreeNode tn2 = new TreeNode();
      tn2.Text = "Fete";

      // Adăugăm nodul în control
      tw.Nodes.Add(tn2);
```

```
        // Adăugăm controlul pe formă
        this.Controls.Add(tw);
    }
```



Crearea ierarhiilor

Fiecare nod într-un **TreeView** poate fi *părinte* pentru alte noduri. Se pot crea ierarhii arborescente cu oricâte niveluri. Modificăm exemplul anterior, prin adăugarea de noduri *child* nodurilor "*Baieti*" şi "*Fete*":

```
// Instanţiem un TreeView
TreeView tw = new TreeView();
// Stabilim poziţia pe formă şi dimensiunea controlului:
tw.Location = new System.Drawing.Point(12, 12);
tw.Size = new System.Drawing.Size(153, 133);

// Creăm un prim nod cu eticheta Baieti
TreeNode tn1 = new TreeNode();
tn1.Text = "Baieti";

// Creăm un nod child pentru Baieti:
TreeNode tn11;
// Referinţa tnn1 indică nodul nou creat (cu eticheta Marius)
tn11 = tn1.Nodes.Add("Marius");

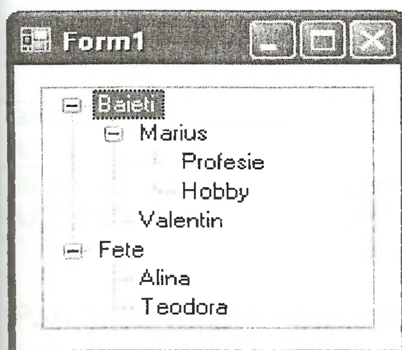
// Nodul Marius va fi părinte pentru alte două noduri:
tn11.Nodes.Add("Profesie");
tn11.Nodes.Add("Hobby");

// Al doilea nod child pentru Baieti:
tn1.Nodes.Add("Valentin");

// Adăugăm nodul Baieti în control:
tw.Nodes.Add(tn1);
```

```
// Creăm al doilea nod rădăcină cu eticheta Fete:
TreeNode tn2 = new TreeNode();
tn2.Text = "Fete";
tn2.Nodes.Add("Alina");
tn2.Nodes.Add("Teodora");

// Adăugăm nodul Fete în control
tw.Nodes.Add(tn2);
// Adăugăm controlul pe formă
this.Controls.Add(tw);
```



Imaginea anterioară reprezintă ceea ce se obţine la rulare.

Important:

Metoda ***Add()*** *returnează întotdeauna o referinţă la nodul nou creat. Aceasta înlesneşte adăugarea de noduri child.*

Accesarea indexată a nodurilor

Subnodurile care fac parte din colecţia **Nodes** a nodului curent se pot accesa cu ajutorul operatorului de indexare. Reluăm exemplul anterior. Prezentăm o alternativă de construire programatică a controlui, bazată pe accesarea indexată:

```
// Instanţiem un TreeView
TreeView tw = new TreeView();

// Stabilim poziţia pe formă şi dimensiunea controlului:
tw.Location = new System.Drawing.Point(12, 12);
tw.Size = new System.Drawing.Size(153, 133);

// Adăugăm în control un prim nod:
tw.Nodes.Add("Baieti");

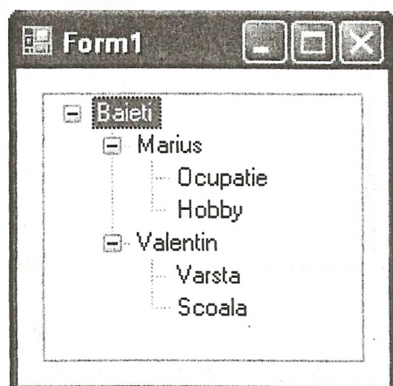
// Adaugăm subnodurile Marius si Valentin pentru Baieti
tw.Nodes[0].Nodes.Add("Marius");
tw.Nodes[0].Nodes.Add("Valentin");

// Adaugăm subnodurile Ocupatie si Hobby pentru Marius:
tw.Nodes[0].Nodes[0].Nodes.Add("Ocupatie");
tw.Nodes[0].Nodes[0].Nodes.Add("Hobby");
```

```
// Adaugăm subnodurile Varsta şi Scoala pentru Valentin:
tw.Nodes[0].Nodes[1].Nodes.Add("Varsta");
tw.Nodes[0].Nodes[1].Nodes.Add("Scoala");

// Adăugăm controlul pe formă:
this.Controls.Add(tw);
```

La execuţie se obţine :



Principalii membri ai clasei TreeView

Controlul **TreeView** are multiple capabilităţi. Descriem cîteva dintre acestea:

- Poate afişa imagini asociate fiecărui nod.
- Poate afişa opţional *checkbox*-uri asociate nodurilor.
- Îşi poate schimba aparenţa setând corespunzător proprietăţi de stil.
- Poate răspunde la diverse evenimente, cum ar fi click sau dublu click pe eticheta unui nod.
- Se poate folosi pentru afişarea informaţiilor dintr-o bază de date, a informaţiilor din fişiere, de pe disc, sau a informaţiilor din aplicaţii.

Aceste calităţi se datorează mulţimii de metode, proprietăţi şi evenimente ale clasei. Dintre acestea menţionăm:

Proprietăţi :

`Nodes`	- Depozitează colecţia de obiecte de tip `TreeNode` a controlului. Fiecare nod are la rândului lui proprietatea `Nodes`, care găzduieşte propriile noduri ***child***.
`LabelEdit`	- Stabileşte dacă eticheta text a unui nod poate fi editată
`CheckBoxes`	- Stabileşte dacă se afişează un checkbox lângă etichetele nodurilor
`SelectedNode`	- Returnează sau modifică nodul selectat.
`PathSeparator`	- Returnează sau setează stringul separator folosit în calea spre noduri. Calea spre un nod este un set de etichete de noduri separate prin delimitatorul `PathSeparator`.

Metode:

`CollapseAll()`	- Colapsează ramurile arborelui (Inversul expandării)
`BeginUpdate()` `EndUpdate()`	- Suspendă redesenarea controlului **TreeView** la fiecare adăugare a unui item în situația în care se adaugă un mare număr de itemi. - Se apelează a doua metodă după terminarea operației de update, pentru a permite controlului să se redeseneze.
`GetNodeAt()`	- Returnează nodul care se găsește la locația specificată

Evenimente:

`BeforeCollapse` `AfterCollapse`	- Se declanşează înainte, respectiv după colapsul unui nod
`BeforeExpand` `AfterExpand`	- Se declanşează înainte, respectiv după expandarea unui nod

Parcurgerea şi prelucrarea nodurilor

Uneori trebuie să vizitați toate nodurile unui **TreeView** pentru o prelucrare oarecare. Amintiți-vă că un **TreeView** este o structură de date arborescentă. Veți înțelege din ce cauză parcurgerea unei asemenea structuri se face cu o metodă algoritmică. Propunem o parcurgere în adâncime:

```
private void ParcurgTreeView(TreeNodeCollection noduri)
{
      foreach (TreeNode nod in noduri)
      {
            PrelucrezNod(nod);
            ParcurgTreeView(nod.Nodes);
      }
}

private void PrelucrezNod(TreeNode nod)
{
      // Prelucrarea dorită pentru nod
      // Aici, pentru verificare afişăm doar eticheta
      MessageBox.Show(node.Text + "\n");
}

private void button1_Click(object sender, EventArgs e)
{
      //
      ParcurgTreeView(treeView1.Nodes);
}
```

De reținut:

- `treview1.Nodes` reprezintă colecția de noduri *rădăcină* în arbore. Fiecare nod din această colecție are la rândul lui proprietatea `Nodes`, care referă nodurile de pe al doilea nivel, etc.
- **TreeNodeCollection** este un tip de date care reprezintă o colecție de obiecte **TreeNode**.

Aplicația *TreeViewExample*

Proiectul pe care îl propunem utilizează un *combobox* pentru selectarea *drive*-urilor sistemului. În funcție de discul selectat, într-un control **TreeView** se afişează structura arborescentă de directoare discului.

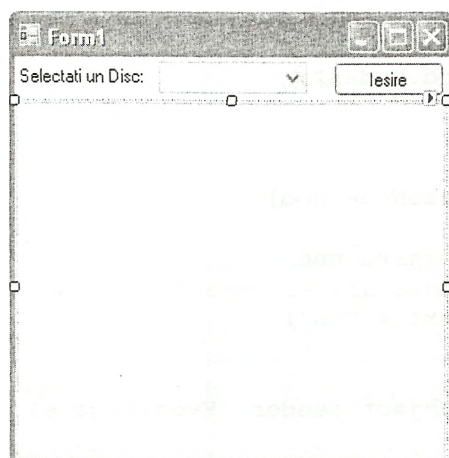
Elemente de noutate:

- Obținerea informațiilor despre discuri cu metoda:
  `DriveInfo.GetDrives()`
- Obținerea numelor subdirectorilor unui director specificat, cu metoda:
  `Directory.GetDirectories()`
- Parcurgerea recursivă a structurii de directoare a unui disc, în scopul populării controlului **TreeView**.

Urmați paşii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *TreeViewExample*.

2. Aduceți din **Toolbox** pe suprafața formei controalele: un **Label**, un **ComboBox**, un **Button** şi un **TreeView** şi aranjați-le astfel:



3. Modificați culorile controlulului `TreeView` după preferință: culoarea fundalului (proprietatea `BackColor`), culoarea etichetelor (`ForeColor`) şi culoarea liniilor punctate (`LineColor`).

4. Adăugați directiva `using System.IO;`

5. În fişierul *Form1.cs*, în corpul clasei **Form1**, scrieți secvențele de cod următoare:

```
public Form1()  // Constructoul clasei
{
    InitializeComponent();
    // Populează comboBox1 cu toate drive-urile
    DriveInfo[] drives = DriveInfo.GetDrives();
    comboBox1.Items.AddRange(drives);
}

 // Câmpuri private. Reprezintă caracterul de separare
 // a căii de director, respectiv litera discului
 private char[] sep = new char[] { '\\' };
 private string drvLetter = null;
 // Deep First în structura de directoare
 private void PopulateTreeView(string path,
                               TreeNode node)
 {
     string[] dirs = Directory.GetDirectories(path);
     string[] auxs;
     foreach (string dir in dirs)
     {
         auxs = dir.Split(sep);
         TreeNode tn =
               new TreeNode(auxs[auxs.Length - 1]);
         node.Nodes.Add(tn); // Adaug subdirectorul
         PopulateTreeView(dir, tn);
     }
 }
```

6. Aplicația trebuie să reacționeze la alegerea unui item în *combobox*. Tratăm evenimentul **SelectedIndexChanged** pentru **comboBox1**. Pentru *aceasta*, acționați dublu click pe acest control. În corpul *handler*-ului de eveniment, scrieți codul evidențiat în bold:

```
private void comboBox1_SelectedIndexChanged(
                           object sender, EventArgs e)
{
   // Obținem textul itemului selectat în combobox
   drvLetter =
   comboBox1.Items[comboBox1.SelectedIndex].ToString();

   // Suspendăm redesenarea controlului
   treeView1.BeginUpdate();

   // Ştergem toată colecția de noduri
   treeView1.Nodes.Clear();
```

```
        // Adăugăm eticheta drive-ului ca nod radacină
        treeView1.Nodes.Add(drvLetter);
        drvLetter += "\\";     // Concatenez caracterul '\'

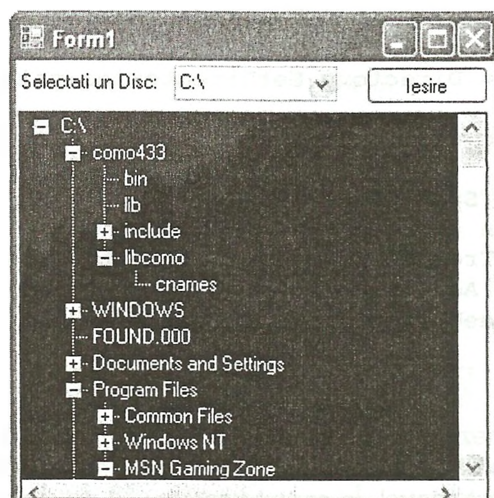
        // Explorarea începe de la rădăcina drive-ului
        PopulateTreeView(drvLetter, treeView1.Nodes[0]);
        treeView1.EndUpdate();
    }
```

7. Acționați dublu click pe butonul *Iesire*. În *handler*-ul evenimentului **Click**, scrieți:

```
Application.Exit();
```

8. Compilați şi rulați cu **F5**.

La rulare obțineți:



Observație:

Cu metoda recursivă se parcurge întreaga structură de director a *drive*-ului curent. Din această cauză, actualizarea controlului **TreeView** este lentă pentru un disc încărcat.

De reținut:

- Clasa `Directory` din spațiul de nume **System.IO** furnizează metode statice pentru creare, copiere, mutarea, redenumire de directori.
- Clasa `DriveInfo` din spațiul de nume **System.IO** conține metode şi proprietăți pentru determinarea informațiilor referitoare la *drive*-uri.
- Metoda `treeView1.Nodes.Clear();` şterge toate nodurile din colecție.

Probleme propuse

1. Creați o aplicație de tip *WindowsForms*. Forma conține un control **TreeView**. Populați *design time* controlul cu noduri.
   Indicație: În fereastra `Properties`, acționați butonul cu eticheta (...) aflat în dreapta proprietății `Nodes`. În dialogul *TreeNode Editor*, adăugați noduri apăsând butoanele *Add Root* şi *Add Child*.

2. Implementați aplicației *TreeViewExample* următoarea facilitate: fiecare nod are lângă etichetă un *checkbox*.

3. * Modificați aplicația *TreeViewExample* de aşa manieră încât nici un nod să nu se adauge în *TreeView* decât în momentul când se expandează părintele său în control. Un nod se expandează prin click pe cruciulița din dreptul său. În felul acesta aplicația va lucra mai rapid.

4. *Implementați un *file browser* cu funcționalitate asemănătoare *Windows Explorer*. Aplicația trebuie să afişeze în panoul din dreapta fişierele aflate în directorul care este selectat în panoul din stânga. De asemenea, la dublu click pe un fişier executabil în panoul din dreapta, acesta să se lanseze în execuție.
   Indicație: Utilizați un **SplitContainer**. În panoul din stânga aduceți un control **TreeView**, iar în cel din dreapta un **ListView**. Pentru obținerea fişierelor dintr-un director oarecare, apelați metoda `Directory.GetFiles()`. Pentru lansare în execuție, utilizați metoda `System.Diagnostics.Process.Start()`.

Controalele Web Browser şi StatusStrip

Dacă v-ați gândit să integrați un *browser Web* în aplicația dumneavoastră, atunci cu *Visual C# Express Edition 2008* aceasta este o sarcină simplă, deoarece mediul include un control de tip **WebBrowser**.

Aplicația *WebBrowserExample*

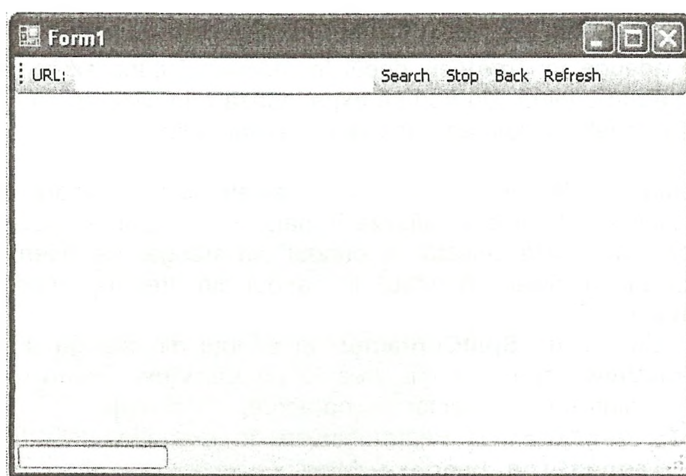
În acest proiect utilizăm un control de tip **WebBrowser**, un control **StatusStrip** pentru vizualizarea încărcării paginiii curente şi un **ToolStrip** pentru butoane şi bara de navigare. Urmați paşii:

1. Creați un nou proiect de tip *Windows Forms* cu numele *WebBrowserExample*.
2. Pe forma aplicației aduceți din **Toolbox** un control de tip **ToolStrip**, un control de tip **StatusStrip** şi un control de tip **WebBrowser**.

3. Selectați controlul **ToolStrip**. Adăugați pe el un **Label**, un **TextBox** și patru butoane. Pentru fiecare buton, acționați click drept și alegeți *DisplayStyle*, apoi *Text*, pentru a afișa text în loc de imagine.

4. Selectați controlul **StatusStrip**. Acționați click drept și alegeți un **ProgressBar**.

5. Selectați controlul **WebBrowser**. În fereastra **Properties** atribuiți proprietății `Dock` valoarea *Fill* și proprietății `URL` adresa: http://www.google.com. Setați textele pentru butoane și aranjați totul astfel:



6. Tratăm evenimentul **Click** pentru butonul *Search*. Acționați dublu click pe buton. în corpul metodei *handler*, scrieți:

```
private void toolStripButton1_Click(object sender,
                                    EventArgs e)
{
    // Încarcă pagina de la adresa specificată
    webBrowser1.Navigate(toolStripTextBox1.Text);
}
```

7. Tratăm evenimentul **Click** pentru butonul *Stop*. Acționați dublu click pe buton. în corpul metodei *handler*, scrieți:

```
private void toolStripButton2_Click(object sender,
                                    EventArgs e)
{
    // Oprește încărcarea paginii
    webBrowser1.Stop();
}
```

8. Tratăm evenimentul **Click** pentru butonul *Back*. Dublu click pe buton. În corpul metodei de tratare, scrieți:

```
private void toolStripButton3_Click(object sender,
                                     EventArgs e)
{
    // Revenire la pagin aanterioară
    webBrowser1.GoBack();
}
```

9. Tratăm evenimentul **Click** pentru butonul *Refresh*. Dublu click pe buton. În corpul metodei de tratare, scrieți:

```
private void toolStripButton4_Click(object sender,
                                     EventArgs e)
{
    // Reîncărcarea paginii de la URL-ul curent
    webBrowser1.Refresh();
}
```

10. În *progress bar*-ul integrat în **StatusStrip** dorim să vizualizăm progresul încărcării paginii. În acest scop, tratăm evenimentul **ProgressChanged** pentru controlul **WebBrowser**. Selectați controlul şi acționați dublu click pe acest eveniment în fereastra **Properties**. Introduceți codul:

```
private void webBrowser1_ProgressChanged(object sender,
                 WebBrowserProgressChangedEventArgs e)
{
    // Setează Maximum la numărul total de bytes pe
    // care îl ocupă documentul care se descarcă
    toolStripProgressBar1.Maximum =
                               (int)e.MaximumProgress;
    // Setează Value la numărul de bytes care au fost
    // downloadaţi până în prezent
    toolStripProgressBar1.Value =
                               (int)e.CurrentProgress;
}
```

11. În momentul în care documentul este descărcat, dorim să ne asigurăm că *progress bar*-ul este la valoarea maximă. Tratăm evenimentul **DocumentCompleted** declanşat de *WebBrowser* când documentul este complet descărcat. Dublu click pe acest eveniment în **Properties**. În metoda de tratare, introduceți codul:

```
private void webBrowser1_DocumentCompleted(
 object sender, WebBrowserDocumentCompletedEventArgs e)
{
    toolStripProgressBar1.Value =
                         toolStripProgressBar1.Maximum;
}
```

12. Compilaţi şi rulaţi cu **F5**.

La rulare, aplicaţia încarcă în mod automat pagina stabilită *design time* prin proprietatea **URL**. În fereastra de navigare puteţi introduce oricarea altă adresă:



Observaţii:

- Dacă lucraţi în **Visual C# Express Edition 2005**, atunci controlul **WebBrowser** nu se găseşte în mod implicit în **Toolbar**. Aceasta nu este o problemă, pentru că el există ca şi componentă **.NET** şi îl puteţi aduce în **Toolbar** cu un click dreapta pe suprafaţa liberă de jos, apoi selectaţi *Chose Items...* iar în tab-ul *.NET Framework Components*, alegeţi **Web Browser**.

- Clasa **WebBrowser** are metode prin care implementaţi acţiunile specifice ale unui browser: navigare, reîncărcare, întoarcere la *URL*-ul anterior şi aşa mai departe.

Integrarea WindowsMediaPlayer în aplicații

Suntem siguri că v-ați întrebat dacă este greu să implementați *player*-e video sau audio. Nu este uşor. Însă există există o cale mai simplă de a avea toate acestea, prin integrarea componentei COM, **WindowsMediaPlayer** în aplicația dumneavoastră.

Ca să vedeți cum se procedează, vom face următorul proiect:

Aplicația *WindowsMediaPlayerExample*

Aplicația utilizează componenta ***Windows Media Player*** şi un control de tip **ToolStrip** cu **5** butoane. Rolul butoanelor este acela de a efectua aceleaşi operații ca şi butoanele implicite ale controlului. Ne vom servi şi de dialogul predefinit **OpenFileDialog** pentru alegerea unui fişier audio sau video de pe disc.
Urmați paşii:

1. Creeați un nou proiect de tip *Windows Forms.*

2. Pe suprafața designerului plasați un control de tip **ToolStrip**, pe care adăugați apoi cinci butoane care afişează text. Etichetați cele cinci butoane astfel: *Open*, *Start*, *Stop*, *Pause*, *Resume*.

3. Plasați pe suprafața formei un control de tip **OpenFileDialog**.

4. ***Windows Media Player*** nu este disponibil în **Toolbox** în mod implicit, dar poate fi adus acolo manual. Îl aduceți cu un click dreapta pe suprafața liberă de jos, apoi selectați *Chose Items...* iar în *tab*-ul *COM Components*, alegeți ***Windows Media Player***. În continuare, plasați componenta pe suprafața *designer*-ului şi setați-i proprietatea `Dock` la valoarea *Fill* şi proprietatea **Name** la valoarea `wmp`.



5. Tratăm evenimentul **Click** pentru butonul *Open*. Acționați dublu click pe buton și scrieți în metoda *handler*:

```
private void toolStripButton1_Click(object sender,
                                     EventArgs e)
{
    // Dacă s-a ales un fişier media şi s-a apăsat OK
    if (openFileDialog1.ShowDialog() ==
                                     DialogResult.OK)
    {
        // Playerul nu va executa play în mod automat
        // la încărcarea fişierului
        wmp.settings.autoStart = false;

        // Se încarcă fişierul ales de utilizator
        wmp.URL = openFileDialog1.FileName;
    }
}
```

6. Tratăm evenimentul **Click** pentru butonul *Start*. Acționați dublu click pe buton și introduceți codul:

```
private void toolStripButton2_Click(object sender,
                                     EventArgs e)
{
     // Player-ul rulează fişierul încărcat
     wmp.Ctlcontrols.play();
}
```

7. Tratăm evenimentul **Click** pentru butonul *Stop*. Dublu click pe buton și introduceți codul:

```
private void toolStripButton3_Click(object sender,
                                     EventArgs e)
{
     // Opreşte rularea
     wmp.Ctlcontrols.stop();
}
```

8. Tratăm evenimentul **Click** pentru butonul *Pause*. Dublu click pe buton. În metoda de tratare introduceți codul:

```
private void toolStripButton4_Click(object sender,
                                     EventArgs e)
{
     // Pauză. La un nou apel play(), redarea continuă
     // din această poziţie în fişier
     wmp.Ctlcontrols.pause();
}
```

9. Tratăm evenimentul **Click** pentru butonul *Resume*. Dublu click pe buton. În metoda de tratare introduceți codul:

```
private void toolStripButton5_Click(object sender,
                                    EventArgs e)
{
    // Redarea se reia din poziția în care s-a ajuns
    // la ultimul apel pause()
    wmp.Ctlcontrols.play();
}
```

10. Compilați și rulați cu **F5**.

Puteți rula atât fișiere audio cât și video.



Observație:

Controalele adăugate manual au aceleași funcții cu cele ale butoanelor implicite. A fost un exercițiu pentru cazul în care vreți să mascați bara standard și să utilizați propriile controale pentru manevrarea player-ului.

Capitolul 8

Desenare în .NET cu Visual C#

GDI+ (*Graphics Device Interface)* este o interfață de programare a aplicațiilor integrată sistemelor de operare *Microsoft Windows XP* şi *Windows Server 2003.*

Aplicațiile de tip Windows desenează elemente grafice cu ajutorul bibliotecii de clase **GDI+**. O interfață pentru dispozitive grafice, aşa cum este **GDI+**, permite programatorilor să deseneze pe dispozitive grafice: ecran, scanner, imprimantă, cu ajutorul aceloraşi funcții, fără să trebuiască să țină seama de detaliile unui dispozitiv de afişare particular.

Se pot desena linii, curbe, figuri geometrice, text, imagini. Clasele **GDI+** se găsesc în marea lor majoritate în spațiul de nume **System.Drawing**. Dintre acestea, clasa **System.Drawing.Graphics** este cea mai utilizată.

Clasa Graphics

Clasa defineşte o suprafață de desenare **GDI+**. Metodele sale desenează pe această suprafață, iar biblioteca **GDI+** ştie să trimită imaginea pe dispozitivul grafic (monitor, imprimantă, sau altceva).

Principalele metode ale clasei Graphics

`Clear()`	- Curăță suprafața de desenare şi o umple cu o culoare de fundal specificată
`DrawArc()`	- Desenează un arc de elipsă
`DrawImage()`	- Desenează o imagine la o locație dată
`DrawLine()`	- Desenează un segment specificat prin capetele sale
`DrawEllipse()`	- Desenează o elipsă specificată printr-un dreptunghi care o mărgineşte
`DrawRectangle()`	- Desenează un dreptunghi
`DrawString()`	- Desenează un text la o locație specificată
`FillEllipse()`	- Umple interiorul unei elipse cu ajutorul unei pensule (obiect de tip **Brush**)
`FillRectangle()`	- Umple interiorul unui dreptunghi cu ajutorul unei pensule

Clasa nu răspunde la evenimente findcă nu are membri de tip **event**. Dar foloseşte evenimentul **Paint** al controalelor pentru a desena pe suprafața lor.

Penițe pentru desenarea formelor

Desenăm cu ajutorul unui obiect de tip **Graphics**. Obiectul se obține în două moduri:

1. Prin intermediul parametrului de tip **PaintEventsArgs** al *handler*-ului evenimentului **Paint**.
2. Cu ajutorul metodei `CreateGraphics()`, moştenită de către toate controalele de la clasa de bază **Control**.

Detaliem cele două cazuri:

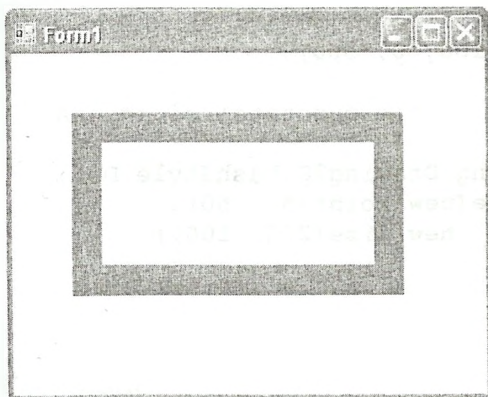
Cazul 1.

Creaţi un proiect nou de tip *Windows Forms*. Tratăm evenimentul **Paint** al formei. *Selectaţi forma şi în fereastra* **Properties** *acţionaţi dublu click pe* evenimentul **Paint**. Scrieţi codul:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
     // Peniţă de culoare roşie şi grosime 20
     Pen p = new Pen(Color.Blue, 20);

     Rectangle r = new Rectangle(new Point(50, 50),
                                 new Size(200, 100));
     e.Graphics.DrawRectangle(p, r);
     p.Dispose();
}
```

Compilaţi şi rulaţi cu **F5**. Pe formă s-a desenat un dreptunghi.



Handler-ului evenimentului **Paint** are un parametru `e` de tip **PaintEventArgs**. Acesta returnează un obiect de tip **Graphics** prin proprietatea `Graphics`: `e.Graphics`.

IMPORTANT !

Pentru desenarea liniilor şi a curbelor, aveţi nevoie de un obiect de tip **Graphics** şi un obiect de tip **Pen** (peniţă). Obiectul **Graphics** furnizează metodele care desenează, iar obiectele de tip **Pen** depozitează atribute ale liniei, cum ar fi culoarea, grosimea şi stilul.

Rețineți că este o practică bună aceea de a elibera resursele deținute de obiectul **Pen**, cu metoda `Dispose()`, în momentul în care nu mai aveți nevoie de el.

Cazul 2.

Metoda `CreateGraphics()`, se regăsește în toate clasele derivate din **Control**. Returnează un obiect de tip **Graphics**. Este nevoie de această metodă când nu doriți să foloiți un eveniment **Paint** pentru a desena.

Realizați un nou proiect de tip *Windows Forms*. Pe formă plasați un buton. Acționați dublu click pe buton pentru tratarea evenimentului **Click**.
În *handler*-ul evenimentului introduceți secvența de cod evidențiată cu **Bold**:

```
private void button1_Click(object sender, EventArgs e)
{
    // Obţinem un obiect Graphics
    Graphics g = this.CreateGraphics();

    // Fabricăm o peniţă
    Pen p = new Pen(Color.Red, 3);

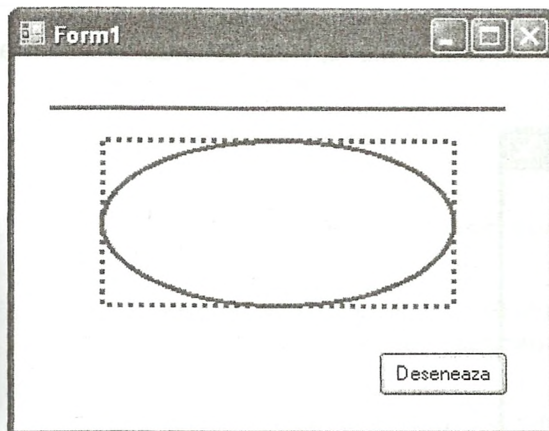
    // Desenăm o linie
    g.DrawLine(p, new Point(20, 30), new Point(280, 30));

    // Desenăm un arc definit de dreptunghiul (50, 50, 200,
    // 100). Unghiul de început e 0, iar cel de sfârşit 360.
    // Aceasta însemnă elipsă
    g.DrawArc(p, 50, 50, 200, 100, 0, 360);

    // Schimbăm stilul peniţei (linie punctată) şi desenăm
    // un dreptunghi
    p.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;
    Rectangle r = new Rectangle(new Point(50, 50),
                                new Size(200, 100));
    g.DrawRectangle(p, r);

    p.Dispose();
}
```

La click pe butonul *Desenează*, pe suprafața formei se desenează figurile:



Observații

- Un arc este o porțiune a unei elipse. Ca să desenați o elipsă, apelați metoda `DrawEllipse()`. Parametrii metodei `DrawArc()` sunt aceeiaşi cu cei ai metodei `DrawEllipse()`, cu diferența că `DrawArc()` necesită un unghi de start şi unul de baleiere. Iată cum desenați o elipsă:

```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Red, 3);
Rectangle r = new Rectangle(new Point(50, 50),
                            new Size(200, 100));
g.DrawEllipse(p, r);
p.Dispose();
```

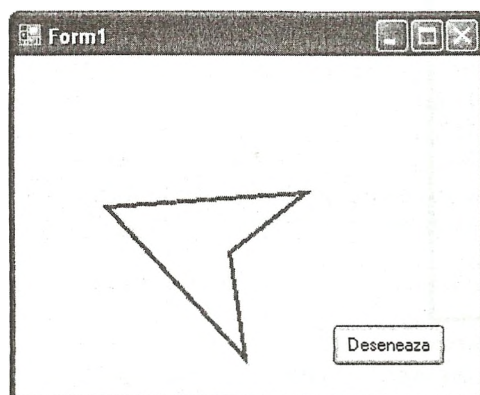
- Metodele de desenare ale arcelor şi elipselor primesc ca parametri coordonatele dreptunghiului care le încadrează. Metodele sunt supraîncărcate. De exemplu, puteți înlocui apelul:
  `g.DrawArc(p, 50, 50, 200, 100, 0, 360);` cu apelul
  `g.DrawArc(p, r, 0, 360);` unde `r` este dreptunghiul definit anterior.

Poligoane

Desenarea unui poligon se face foarte simplu, prin apelul metodei `DrawPolygon()`. Metoda are ca parametri un obiect de tip `Pen` şi un tablou de puncte:

```
Graphics g = this.CreateGraphics();
Pen p = new Pen(Color.Red, 3);
Point[] pt = { new Point(190, 90), new Point(140, 130),
               new Point(150, 200), new Point(60, 100) };
// Desenează un poligon cu patru vârfuri
g.DrawPolygon(p, pt);
p.Dispose();
```

Dacă ne folosim de evenimentul Click al butonului în aplicația anterioară, atunci la click pe *Desenează*, avem:



Pensule pentru umplerea formelor

Figurile închise se desenează cu penițe. Interiorul lor se umple cu pensule. Pensulele sunt obiecte de tip **Brush**.

Biblioteca **GDI+** oferă câteva clase care definesc pensule: **SolidBrush**, **HatchBrush**, **TextureBrush**, **LinearGradientBrush**, și **PathGradientBrush**.

Clasa **SolidBrush** este definită în spațiul de nume **System.Drawing**. Celelalte pensule sunt în spațiul de nume **System.Drawing.Drawing2D.**
Cel mai simplu mod de a învăța, este de a lucra cu aceste pensule. Vom realiza o aplicație ca model.

Aplicația *BrushesExample*

Aplicația umple trei elipse cu pensule diferite: o pensulă solidă, una de hașurare și una gradient liniar. În dreapta formei se va umple un dreptunghi cu o pensulă de tip **TextureBrush**, care utilizează ca element de umplere o imagine.

Urmați pașii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *BrushesExample*.

2. În fișierul *Form1.cs*, în clasa `Form1`, introduceți la secțiunea directivelor **using**, secvența:

```
using System.Drawing.Drawing2D;
```

3. Tratăm evenimentul **Paint** pentru formă. Selectați forma. În fereastra **Properties**, acționați dublu click pe evenimentul **Paint**. În corpul *handler*-ului introduceți codul:

```
private void Form1_Paint(object sender,
                              PaintEventArgs e)
{
    Pen p = new Pen(Color.Red, 3);

    // Creăm un dreptunghi pentru elipse
    int x = 10, y = 10, width = 200, height = 100;
    Rectangle r = new Rectangle(x, y, width, height);

    // Creăm o pensulă solidă
    SolidBrush s = new SolidBrush(Color.Aquamarine);

    // Umplem elipsa cu pensula solidă
    e.Graphics.FillEllipse(s, r);

    y = 120;
    r = new Rectangle(x, y, width, height);
    // Creăm o pensulă de haşurare
    HatchBrush h =
              new HatchBrush(HatchStyle.DiagonalCross,
                             Color.Azure, Color.Black);

    // Umplem elipsa cu pensula de haşurare
    e.Graphics.FillEllipse(h, r);

    y = 230;
    r = new Rectangle(x, y, width, height);

    // Creăm o pensulă gradient liniar
    LinearGradientBrush lg = new LinearGradientBrush(r,
                        Color.Aqua, Color.BlueViolet,
                        LinearGradientMode.Horizontal);

    // Umplem elipsa cu pensula gradient liniar
    e.Graphics.FillEllipse(lg, r);

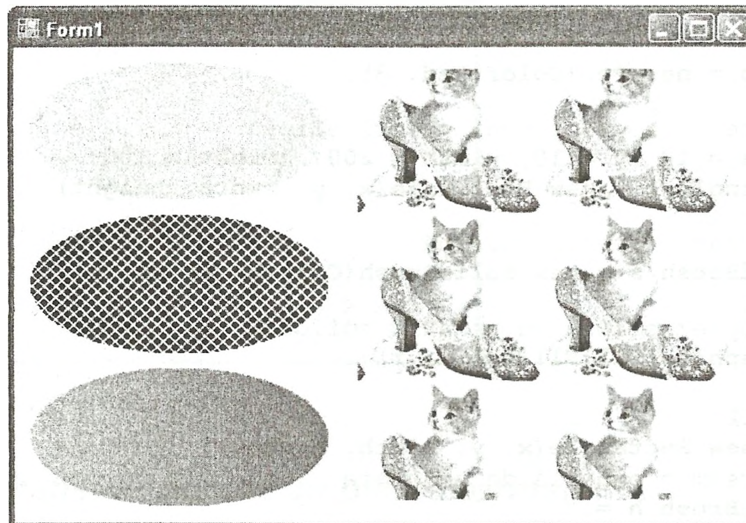
    x = 230; y = 15; width = 250; height = 310;
    r = new Rectangle(x, y, width, height);

    // Creăm o pensulă de textură
    TextureBrush t =
       new TextureBrush(Image.FromFile("pisica.jpg"));

    // Umplem dreptunghiul cu pensula de textură
    e.Graphics.FillRectangle(t, r);
    p.Dispose();
}
```

4. Compilați, rulați cu **F5**.

La rulare, avem:



Observații:

- Constructorii claselor de tip *brush* primesc ca argumente culori, stiluri, etc, prin intermediul membrilor anumitor enumerări. Ați remarcat enumerările **Color**, **LinearGradientMode**, **HatchStyle**.
- Constructorul `TextureBrush((Image.FromFile("pisica.jpg"));` încarcă o imagine. Dacă se specifică doar numele fişierului imagine, atunci acesta este căutat în folderul în care se găseşte fişierul *assembly*, adică **/bin/Debug** sau **/bin/Release**.

Desenarea textului

GDI+ furnizează câteva clase care se ocupă cu desenarea textului. Clasa **Graphics** are în acest scop mai multe metode `DrawString()`. Vă veți acomoda cu utilizarea acestora, urmărind exemplul de mai jos:

Aplicația *DrawStringExample*

Ca să desenați cu metodele `DrawString()` trebuie să pregătiți un font şi o pensulă. Aplicația testează două dintre metodele `DrawString()`. Prima scrie un text la o locație specificată de un punct, iar a doua scrie un text încadrat într-un dreptunghi.

Urmați pașii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *DrawStringExample*.

2. Tratăm evenimentul **Paint** pentru formă. Selectați forma. În fereastra **Properties**, acționați dublu click pe evenimentul **Paint**. În corpul *handler*-ului introduceți codul:

```
private void Form1_Paint(object sender,
                                       PaintEventArgs e)
{
    String s = "Dau un regat pentru un cal";

    // Creăm un font precizând familia, dimensiunea
    // şi stilul, apoi o pensulă solidă
    Font f = new Font("Arial", 22, FontStyle.Bold);
    SolidBrush b = new SolidBrush(Color.Red);

    // Punctul colţului stânga sus al stringului
    PointF pt = new PointF(20.0F, 30.0F);

    // Desenează stringul
    e.Graphics.DrawString(s, f, b, pt);

    float x = 100.0F, y = 100.0F, width = 200.0F,
          height = 80.0F;
    RectangleF r = new RectangleF(x, y, width, height);

    // Desenează dreptunghiul pe ecran
    Pen p = new Pen(Color.Black);
    e.Graphics.DrawRectangle(p, x, y, width, height);

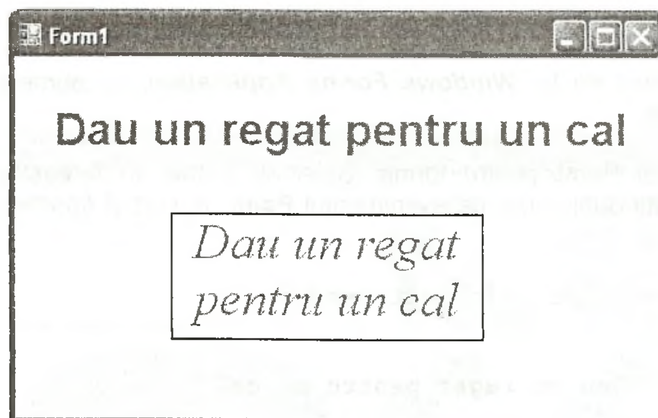
    // Formatăm stringul. Aliniere centru
    // (in dreptunghi)
    StringFormat stF = new StringFormat();
    stF.Alignment = StringAlignment.Center;
    f = new Font("Times New roman", 24,
                                FontStyle.Italic);

    // Desenez textul in dreptunghiul r
    e.Graphics.DrawString(s, f, b, r, stF);

    p.Dispose();
}
```

3. Compilați și rulați cu **F5**.

La rulare, obțineți:



Observații:

- Clasa **StringFomat** are proprietatea `Alignment`, care permite să setați alinierea textului în dreptunghiul specificat.
- `PointF` și `RectangleF` sunt structuri similare `Point` și `Rectangle`, cu diferența că admit coordonate reale.
- A doua metodă `DrawString()` constrânge stringul să se încadreze în dreptunghiul specificat și îl trunchiază în așa fel încât cuvintele individuale nu se frâng.

De reținut:

- ✓ Cu ajutorul bibliotecii **GDI+** se scriu aplicații care desenează pe dispozitive grafice (ecran, imprimante).
- ✓ Clasa **System.Drawing.Graphics** este nucleul metodelor de desenare.
- ✓ Pentru a desena o curbă, aveți nevoie de o peniță (obiect de tip **Pen**), iar pentru a umple sau hașura o curbă închisă, aveți nevoie de o pensulă (obiect de tip **Brush**).
- ✓ Obiectele de tip **Graphics**, cu care apelați metodele **Draw**, se obțin în două moduri:
    1. Din parametrul de tip **PaintEventArgs** al metodei de tratare a evenimentului **Paint**.
    2. Prin apelul metodei `CreateGraphics()`.
- ✓ Textul se desenează cu metodele **DrawString()** ale clasei **Graphics**.

Probleme propuse

1. Să presupunem că în *handler*-ul evenimentului **Paint** desenați pe formă un text. Stringul care trebuie desenat este itemul pe care s-a făcut click într-un **Listbox**. Cum forțați redesenarea formei (declaşarea evenimentului **Paint**) la click pe acel item ? Realizați o aplicație care implementează această cerință.
   Indicație: În *handler*-ul eveniementului **SelectedIndexChanged** al controlului **ListBox**, preluați textul necesar şi foțați evenimentul **Paint** cu metoda `this.Invalidate()`. Aceasta invalidează toată suprafața formei şi obligă redesenarea.

2. Există patru versiuni ale metodei `DrawEllipse()`. Testați-le pe toate într-un mic proiect.

3. Realizați un proiect în care veți desena icon-uri cu ajutorul celor două metode `DrawIcon()` şi veți afişa imagini, utilizînd câteva versiuni diferite ale metodei `DrawImage()`.

4. Utilizați metoda `Graphics.RotateTransform()`, pentru a roti un string-uri şi figuri geometrice care au fost desenate cu metodele `Draw()`.

5. Realizați un mic proiect care utilizează metodele `DrawPie()`, `DrawPolygon()`, `FillPie()` şi `FillPolygon()`.

Capitolul 9

XML cu C#

XML (***eXtensible Markup Language***) este un limbaj folosit pentru descrierea datelor. Menirea XML este de a oferi un format standard, cu ajutorul căruia aplicaţii diferite, rulând pe calculatoare diferite, pot să citească datele, să le procese şi să le scrie. Sistemele de calculatore şi bazele de date conţin date în formate incompatibile. Datele XML se depozitează în format text. Aceasta a oferit o cale de a depozita şi de a transmite datele în mod independent de software-ul şi hardware-ul folosit.

XML specifică date, dar şi forma în care datele sunt organizate. Formatul XML este folosit pentru depozitarea informaţiilor din documente care conţin cuvinte, pentru menţinerea listelor de preţuri pe site-urile Web, detaliile *post*-urilor de pe bloguri. XML este vehiculul prin care se trimit cantităţi mari de informaţie prin Internet. Datele schimbate între serviciile *Web* şi aplicaţiile clienţilor sunt XML.

Sintaxa XML

Spre deosebire de HTML, XML este destinat depozitării şi transportului datelor şi nu afişării datelor. Tag-urile XML nu sunt predefinite. Trebuie să definiţi dumneavoastră aceste tag-uri. Un fişier XML este un fişier cu text.

```
<?xml version="1.0" encoding="utf-8" ?>
<biblioteca>
     <carte nrvol ="12">
          <titlu>Oameni si soareci</titlu>
          <autor>John Steinbeck</autor>
     </carte>
     <carte nrvol ="7">
          <titlu>Darul lui Humboldt</titlu>
          <autor>Saul Bellow</autor>
     </carte>
</biblioteca>
```

Elementele sunt ceea ce scrieţi între parantezele unghiulare. Exemplu: `biblioteca`, `carte`, `titlu`, `autor`, sunt elemente. Primul element, `<?xml version="1.0" encoding="utf-8" ?>` indică versiunea XML cu care este conformă acest document şi că e conformă cu standardul de codificare *Unicode* UTF-8.
Atributele descriu elementele.

Sintaxa pentru un element este:

```
<nume_element nume_atribut ="valoare_atribut">
       Conţinutul elementului
</nume_element>
```

De exemplu, **12** şi **7** sunt atributele elementelor **carte**. Atributele pot să lipsească. **Comentariile** în document se scriu astfel: **`<!--Comentariu-->`**

Criteriile pe care trebuie să le îndeplinească un document valid XML sunt:

- Documentul are exact un singur element **root**. În cazul de faţă, elementul root este **`biblioteca`**. Acesta îndeplineşte rolul de container pentru restul datelor.
- Pentru fiecare element "start" trebuie să existe un element corespunzător "final". Exemplu: **`<carte>`** şi **`</carte>`**.
- Elementele nu se pot suprapune. Documentul următor este invalid:

```
<biblioteca>
   <carte>
        <titlu>Iarna vrajbei noastre</carte>
   </titlu>
</biblioteca>
```

Clase .NET pentru Xml

Clasele .NET care lucrează cu XML se găsec în spaţiul de nume **System.Xml**. Documentele XML au o structură arborescentă. Trebuie să conţină întotdeauna un unic element **root**. Acesta la rândul lui, conţine atribute, valori, alte elemente, şi aşa mai departe. Clasa **XmlDocument** se bazează pe aceeaşi idee. Un document se obţine prin instanţierea clasei **XmlDocument**. Obiectul are proprietatea **ChildNodes**, care este o colecţie referinţe la nodurile *child*. Fiecare element al colecţiei este un obiect de tip **XmlNode**. Fiecare obiect de tip **XmlNode**, are la rândul lui proprietatea **ChildNode**. Astfel, deoarece în fiecare nod se menţin referinţe spre nodurile child. se poate explora întregul document.

Citirea informaţiilor dintr-un document XML

Un document XML poate îndeplini rolul unei baze de date pentru aplicaţia dumneavoastră. Aplicaţiile citesc deseori informaţii din asemenea fişiere XML. Este important, pentru că dacă doriţi să modificaţi acele informaţii, schimbaţi conţinutul fişierului XML şi nu e nevoie de refaceţi aplicaţia.

Aplicaţiile .NET pot să şi scrie în fişiere .xml. De exemplu, dacă setările pe care le face utilizatorul în aplicaţie trebuie fie persistente, atunci e o idee bună să le salvaţi într-un document XML.

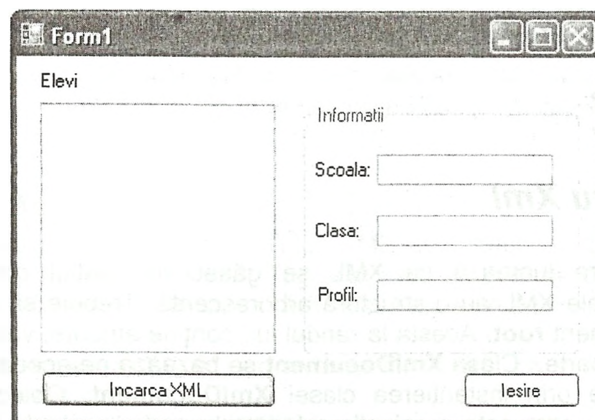
Vom lucra în continuare cu clasa **XmlDocument**, pentru citirea datelor dintr-un document (fişier) XML.

Aplicația *XmlExample*

Aplicația citeşte date din fişierul `Elevi.xml` şi le afişează în mai multe controale. Într-un control de tip **ListBox** se afişează numele elevilor, iar în alte trei controale de tip **TextBox**, se afişează şcoala, clasa şi profilul clasei pentru elevul care a fost selectat în *list box*.
Urmați paşii:

1. Creați un nou proiect de tip ***Windows Forms Application***, cu numele *XmlExample*.

2. Aduceți pe forma aplicației un control de tip **ListBox**, trei controale de tip **TextBox**, două butoane, un **GroupBox**, patru controale **Label** şi aranjați-le astfel:



3. Vom edita fişierul XML. Mediul integrat include un editor de fişiere XML. Acesta se invocă acționând click drept în ***Solution Explorer*** pe titlul proiectului, apoi *Add,* şi *New Item...* În dialogul care apare, alegeți din panoul *Template*, iconul *XML File*. Completați câmpul *Name* cu *Elevi.xml*. În editor scrieți următorul document:

```
<?xml version="1.0" encoding="utf-8" ?>
<absolventi>
    <elev>
        <nume>Ardelean Eugen</nume>
        <scoala>C.N. Liviu Rebreanu</scoala>
        <clasa>a XII-a A</clasa>
        <profil>matematica-informatica</profil>
    </elev>

    <elev>
        <nume>Costea Andrei</nume>
        <scoala>C.N. Andrei Muresanu</scoala>
        <clasa>a IX-a C</clasa>
        <profil>stintele naturii</profil>
    </elev>
```

```
    <elev>
         <nume>Nicoara Alina</nume>
        <scoala>Gr. Sc. Industrial nr 1</scoala>
        <clasa>a XI-a B</clasa>
        <profil>telecomunicatii</profil>
    </elev>
</absolventi>
```

După ce ați editat și ați salvat, copiați fișierul *Elevi.xml* din folderul proiectului, în folderul **/bin/Debug**.

4. În secțiunea se directive a fișierului fișierul *Form1.cs*, introduceți :

```
using System.Xml;
```

5. Declarați două câmpuri private în clasa *Form1*:

```
// d reține întregul document XML
private XmlDocument d = null;

// elevi reține colecția de noduri child a
// nodului curent.
private XmlNodeList elevi = null;
```

6. Tratăm evenimentul **Click** pentru butonul *"Incarca Xml"*. În fereastra **Properties** acționați dublu click pe buton. În corpul *handler*-ului introduceți codul:

```
private void button1_Click(object sender, EventArgs e)
{
    d = new XmlDocument();  // Creează un document XML
    d.Load("Elevi.xml");    // Încarcă fișierul

    // Returnează colecția de elemente (noduri) "elev"
    elevi = d.SelectNodes("absolventi/elev");

    // Parcurge elementele "elev"
    for (int i = 0; i < elevi.Count; i++)
    {
        // În colecția elevi, nodurile se accesează
        // indexat. Pentru fiecare elev, se selectează
        // nodul "nume"
        XmlNode elev =
               elevi.Item(i).SelectSingleNode("nume");

        // Numele elevului se adaugă în listă
        listBox1.Items.Add(elev.InnerText);
    }
}
```

7. La selectarea unui nume în listă, dorim să afișăm în controalele **TextBox** informațiile referitoare la acel nume.

Tratăm evenimentul **SelectedIndexChanged** pentru **ListBox**. Selectați controlul şi în fereastra **Properties**, faceți dublu click pe eveniment. În metoda de tratare, introduceți codul evidențiat:

```
private void listBox1_SelectedIndexChanged(
                            object sender, EventArgs e)
{
    // Se obţine indexul numelui selctat
    int index = listBox1.SelectedIndex;

    // Se selectează lista de elemente "elev"
    elevi = d.SelectNodes("absolventi/elev");

    // În colecţia elevi se accesează cel cu indexul
    // index. Apoi se selectează subelementul "scoala"
    XmlNode nume =
        elevi.Item(index).SelectSingleNode("scoala");

    // Textul aferent elementului "scoala"
    textBox1.Text = nume.InnerText;

    // Operaţiile se repetă pentru "clasa" şi "profil"
    XmlNode clasa =
        elevi.Item(index).SelectSingleNode("clasa");
    textBox2.Text = clasa.InnerText;
    XmlNode profil =
        elevi.Item(index).SelectSingleNode("profil");
    textBox3.Text = profil.InnerText;
}
```

8. Compilați şi rulați cu **F5**.

La rulare, după acționarea butonului *Incarca XML*:

Important!

- Clasele `XmlDocument` şi `XmlNodeList` moştenesc clasa `XmlNode`. Astfel ne explicăm faptul că proprietăţile `ChildNodes` şi `InnerText` se regăsesc printre membrii ambelor clase.
- `InnerText` este valoarea nodului concatenată cu valorile tuturor copiilor săi.
- `XmlNodeList` reprezintă o colecţie ordonată de noduri.
- Metoda `XmlNode Item(int i)` a clasei `XmlNodeList` returnează nodul aflat la indexul `i`.

Descărcarea fişierelor XML de pe Internet

În ultimii ani, multe site-uri şi bloguri expun clienţilor un format XML cunoscut sub numele **RSS** (*Really Simple Syndication*), în scopul distribuirii de conţinut care se schimbă frecvent (ştiri, sumarul conţinutului unui site sau întregul conţinut, posturi pe blog, etc).

Un furnizor de conţinut (site, blog) publică de regulă un link (*feed link*) la care utilizatorii subscriu cu ajutorul unor programe specializate (*feed reader*). Subscrierea se face simplu, copiind link-ul din *Web browser* în *feed reader*. Mai departe, conţinuturile noi se descarcă în mod automat, iar utilizatorul este înştiinţat când acest lucru se petrece. Practic, se descarcă fişiere XML.

Cu .NET puteţi crea propriul *feed reader* (agregator). Nu vom face acest lucru aici, ci vom arăta cât de simplu puteţi descărca un fişier XML de pe Internet sau un *feed* **RSS**.

Aplicaţia *DownloadXml*

Aplicaţia utilizează clasa `WebClient` din spaţiul de nume **System.Net** pentru a descărca un fişier de tip XML.

Urmaţi paşii:

1. Creeaţi o aplicaţie de tip *Windows Forms* în Visual C# Express 2008.

2. Pe forma aplicaţiei, aduceţi din **Toolbox** un buton.

3. La secţiunea directivelor, în fişierul Form1.cs, adăugaţi directivele:

```
using System.Xml;
using System.Net;
```

4. Tratăm evenimentul **Click** generat la click pe buton. Acţionaţi dublu click pe buton. În metoda de tratare scrieţi:

```
private void button1_Click(object sender, EventArgs e)
{
    // Descărcăm în stringul xml conţinutul XML (RSS
    // feed) de la un anumit URL
    string xml = new
    WebClient().DownloadString("http://news.google.com/"
                                + "?output=rss");

    // Creăm o instanţă de tip XmlDocument
    XmlDocument doc = new XmlDocument();

    // Încărcăm stringul (cu conţinut XML) în document
    doc.LoadXml(xml);

    // Salvăm pe disc conţinutul XML descărcat
    doc.Save("continut.xml");
}
```

5. Compilaţi şi rulaţi cu **F5**.

După executare, veţi găsi în folderul **/Bin/Debug** fişierul **continut.xml**.

Observaţie:

Pe numeroase site-uri şi blog-uri veţi găsi *feed link*-uri pentru subscriere. Iată câteva link-uri alese aleatoriu:

```
http://www.preferredjobs.com/rss/rss2.asp
http://sourceforge.net/export/rss2_sfnews.php?feed
http://newsrss.bbc.co.uk/rss/newsonline_uk_edition/world/rss.
xml
```

Citirea şi analiza unui document XML cu XmlTextReader

Transportul datelor la distanţă se face în mod frecvent cu ajutorul *stream*-urilor.

Clasa **XmlTextReader** este un instrument foarte rapid de accesare *stream*-urilor de date XML. Clasa reprezintă un *cititor* (*reader*), care se deplasează în *stream* doar înainte (*forward only*), citeşte documentul nod după nod şi nu modifică *stream*-ul. Reader-ul vă permite să vă deplasaţi progresiv în documentul XML şi să examinaţi fiecare nod, cu elementul, atributele şi valoarea sa.

Propunem o aplicaţie care ilustrează o parte dinte capabilităţile acestei clase.

Aplicația *XmlTextReaderExample*

Programul C# de mai jos folosește clasa **XmlTextReader** pentru a citi un document XML aflat pe disc. Documentul este citit nod cu nod, apoi este reconstituit și se afișează pe ecran.
Urmați pașii:

1. În *Visual C#* creeați un proiect de tip consolă cu numele *XmlTextReaderExample*.

2. Editați fișierul *carti.xml* cu următorul conținut:

```
<!--Citire fisier XML-->
<biblioteca>
  <carte ISBN="186-383-537">
    <Titlu>Amintiri din copilarie</Titlu>
    <Pret>13.20</Pret>
    <Autor>Ion Creanga</Autor>
  </carte>
</biblioteca>
```

3. În fișierul *Program.cs* introduceți codul:

```
using System;
using System.Xml;

class Program
{
   static void Main(string[] args)
   {
       // Creăm un reader (instanță a clasei) și
       // încărcăm fișierul
       XmlTextReader reader =
                    new XmlTextReader ("carti.xml");

      // Citim pe rând toate nodurile din document
      while (reader.Read())
      {
          // În funcție de tipul nodului curent
          switch (reader.NodeType)
          {
              // Dacă nodul este un element
              case XmlNodeType.Element:
                  Console.Write("<" + reader.Name);
                  Console.WriteLine(">");
                  break;

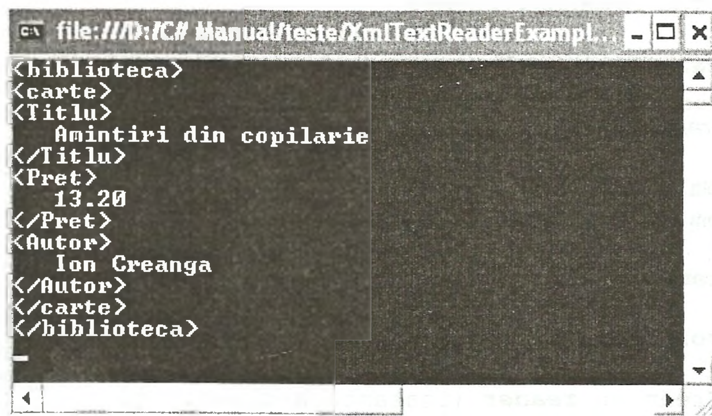
              // Afișăm textul din fiecare element
              case XmlNodeType.Text:
```

```
                    Console.WriteLine ("    " +
                                        reader.Value);
                    break;

                // Afişăm "sfârşitul" elementului
                case XmlNodeType.EndElement:
                    Console.Write("</" + reader.Name);
                    Console.WriteLine(">");
                    break;
            }
        }
    } // Main()
}
```

4. Compilaţi şi rulaţi cu **F5**.

La rulare, pe ecran se va afişa:



De reţinut:

- Metoda **`Read()`** a clasei **`XmlTextReader`** citeşte şi returnează următorul nod din *stream*.
- Proprietatea **`NodeType`** a clasei **`XmlNode`** este o enumerare ai cărei membri reprezintă tipurile de noduri din document. Studiaţi această enumerare pentru a vedea că există mai multe tipuri de noduri decât în acest exemplu.
- Cu ajutorul clasei **`XmlTextReader`** obţineţi conţinutul XML dintr-un fişier sau dintr-un *stream*. În plus, are calitatea că vă permite să analizaţi şi să prelucraţi acest conţinut.

Crearea conținutului XML cu XmlTextWriter

Clasa **XmlTextWriter** oferă o cale rapidă cu care puteți genera fişiere sau *stream*-uri cu conținut XML. Clasa conține un număr de metode şi proprietăți cu care se generează conținut XML. Pentru a le folosi creați un obiect de tip **XmlTextWriter**, apoi adăugați pe rând entități XML obiectului. Există metode cu care puteți adăuga orice tip de conținut XML. Aplicația pe care urmează utilizează câteva dintre aceste metode.

Aplicația *XmlTextWriterExample*

Vom crea un fişier cu conținut XML. Urmați paşii:

1. Creeați o aplicație de tip consolă în *Visual C# Express Edition*.

2. În fişierul *Program.cs*, introduceți codul:

```
using System;
using System.IO;
using System.Xml;

public class Program
{
   // Fişierul în care se scrie conţinutul XML
   private string filename = "continut.xml";

    public static void Main()
    {
       // Creăm obiectul de tip XmlTextWriter
       XmlTextWriter writer =
                     new XmlTextWriter(filename, null);

       // Folosim indentarea tag-urilor
       writer.Formatting = Formatting.Indented;

       // Scriem un comentariu XML
       writer.WriteComment("Creeare fisier XML");

       // Scriem primul element (root).
       writer.WriteStartElement("librarie");

       // Scriem elementul carte(child  pentru
       // librarie)
       writer.WriteStartElement("carte");
       // Scriem un atribut pentru carte
       writer.WriteAttributeString("ISBN",
                                   "186-383-523");
```

```
            // Titlul cărţii
            writer.WriteStartElement("Titlu");
            writer.WriteString("Amintiri din copilarie");
            writer.WriteEndElement();

            // Preţul si autorul cărţii
            writer.WriteElementString("Pret", "13.20");
            writer.WriteStartElement("Autor");
            writer.WriteString("Ion Creanga");

            // Scrie tag-ul de sfârşit pentru elementul
            // carte
            writer.WriteEndElement();

            // Scrie tag-ul de sfârşit pentru elementul
            // librarie
            writer.WriteEndElement();

            // Scrie XML în fişier si închide writer
            writer.Flush();
            writer.Close();

            // Afişăm continutul XML si pe ecran
            XmlDocument doc = new XmlDocument();

            // Păstrăm spaţiile albe pentru lizibilitate
            doc.PreserveWhitespace = true;
            // Încărcăm fişierul
            doc.Load(filename);

            //Scriem conţinutul XML pe ecran
            Console.Write(doc.InnerXml);

            Console.ReadLine();
        } // Main()
    }
```

3. Compilaţi şi rulaţi cu F5.

În folderul **/bin/Debug** veţi găsi fişierul *continut.xml*. Acelaşi conţinut se afişează şi pe ecran:

file:///D:/C# Manual/teste/WriteXML/WriteXML/bin/De...

```
<!--Creeare fisier XML-->
<librarie>
  <carte ISBN="186-383-523">
    <Titlu>Amintiri din copilarie</Titlu>
    <Pret>13.20</Pret>
    <Autor>Ion Creanga</Autor>
  </carte>
</librarie>
```

De reținut:

- Clasa `XmlTextWriter` oferă o cale programatică de a crea conținut XML
- Metodele clasei au nume intuitive.
  De exemplu, `writer.WriteStartElement("titlu");` scrie în obiectul writer `<titlu>`.
- Cu `WriteString()` scrieți textul asociat elementului curent.
- Există și alte metode și proprietăți corespunzătoare altor entități XML. Acestea se utilizează în mod similar. Merită efortul să le explorați.

Probleme propuse

1. Realizați o aplicație care utilizează controale *text box* pentru a introduce datele personale ale clienților unei bănci. Datele clienților se vor salva într-un fișier în format XML.

2. Realizați o aplicație care deschide un fișier XML și afișează într-un *textbox* concatenarea tuturor textelor asociate elementelor *child* ale elementului *root*.

3. Realizați o aplicație de tip consolă care descarcă un *feed* **RSS** de pe Internet și listează numele tuturor elementelor din conținutul XML.

4. Realizați o aplicație care citește conținutul XML dintr-un fișier și cu ajutorul clasei `XmlTextReader` și afișează într-un **ListBox** toate elementele XML.

5. Realizați o aplicație care salvează într-un fișier cu conținut XML informații despre produsele unui magazin. La repornirea aplicației, aceste informații se pot încărca de către aplicație pentru a fi afișate în controale.

Partea a III – a

Baze de date

Scopul acestei părţi a lucrării este acela de a vă ajuta să accesaţi bazele de date relaţionale cu C#.

Capitolul 10

Baze de date şi ADO.NET. Noţiuni introductive

Capitolul are următoarele obiective:

- O scurtă prezentare a sistemului de gestiune pentru baze de date reaţionale ***MS SQL Server***.
- Generalităţi despre sistemele de gestiune a bazelor de date relaţionale.
- Primele noţiuni despre tehnologia **ADO.NET**.
- Crearea unei baze de date în *Visual C# 2008 Express Edition*.
- Interogarea bazei de date cu ajutorul *Query Designer*.

Instrumente de lucru

Instrumentele software pe care le vom utiliza în lucrul cu bazele de date sunt:

- ***Visual C# Express Edition 2008*** (**VCSE**).
- ***Microsoft SQL Server Express Edition 2005*** (**SSE**).

Ambele instrumente funcţionează pe platforma ***Microsoft .NET*** versiunea **3.5**. Sunt versiuni *free*, puternice, destinate să lucreze împreună. În momentul când instalaţi **VCSE**, aveţi opţiune de instalare şi pentru **SSE**.

SSE, este un subset al serverului de baze de date *Microsoft SQL Server 2005*. Acesta din urmă este unul dintre cele mai avansate sisteme de gestiune a bazelor de date (*SGDB*) existente. Deşi SSE nu include toate capabilităţile ***SQL Server 2005***, este perfect funcţional şi utilizabil în aplicaţii industriale. Suportă procesarea online a tranzacţiilor (OLAP), poate să gestioneze baze de date mai mari de **4 Gb** şi admite sute de conexiuni concurente.

Când instalaţi **SSE**, veţi avea de fapt două versiuni ale serverului: ***SQL Server Express Edition*** şi ***SQL Server Express Compact Edition 2005.*** Acesta din urmă, este o versiune cu mai puţine facilităţi, fiind destinat să gestioneze baze de date locale. În momentul în care construiţie o bază de date în VCSE, aveţi opţiunea de a alege unul dintre cee două servere.

SGDB-uri – noţiuni generale

Sistemele de gestiune ale bazelor de date, aşa cum numele indică, se ocupă cu crearea şi întreţinerea bazelor de date. O bază de date este o colecţie de informaţii structurate. Bazele de date poate înmagazina mari cantităţi de informaţie care poate apoi să fie accesată şi modificată în mod eficient de către utilizatori prin intermediul SGDB-urilor.

Un SGDB admite multiple baze de date. Utilizatorii obţin informaţii din bazele de date scriind interogări (*query*) care se adresează serverului de baze de date. Acesta le execută, efectuând operaţii specifice asupra bazei de date şi întoarce utilizatorului un set de rezultate. Operaţiile obişnuite sunt: de modificare a datelor, de introducere de noi date, de ştergere.

SQL (*Structured Query Language*) este limbajul cu care trebuie să ne adresăm unui SGDB care operează cu baze de date relaţionale. În această lucrare vom presupune că cititorii au cunoştinţe elementare referitoare la acest limbaj.

Un ***Sistem de Gestiune a Bazelor de Date Relaţionale*** (SGDBR) este un SGDB bazat pe ***modelul relaţional***. Datele sunt depozitate în tabele. Între tabele există relaţii, iar relaţiile se memorează de asemenea în baza de date.

Calităţile SGDBR-urilor

Sistemele de gestiune a bazelor de date asigură depozitarea şi regăsirea datelor. În raport cu alte forme de păstrare a datelor, au *atu*-uri care le fac inegalabile:

- **Viteza mare a operaţiilor.**
  Operaţiile asupra datelor (căutare, sortare, modificare) sunt foarte rapide.

- **Compactează informaţia.**
  Bazele de date lucrează cu cantităţi uriaşe de date. Acestea se memorează pe spaţii relativ reduse.

- **Asigură securitatea datelor.**
  Datele sunt foarte bine protejate împotriva oricărui acces neautorizat.

- **Întreţine datele.**
  Un SGDB ţine evidenţa fiecărui fragment de informaţie. Utilizatorii nu trebuie să se preocupe acest aspect.

- **Controlează redundanţa datelor.**
  Previn crearea de duplicate multiple ale aceloraşi date, care ar duce la ocuparea unui volum mare pe disc.

- **Previne inconsistenţa datelor.**

Nu permite utilizatorului operații care distruge logica bazei de date. În același timp, când un SGDB elimină date redundante, această operație se face cu păstrarea consistenței datelor.

- **Asigură atomicitatea operațiilor.**
  Operațiile elementare, dar și grupurile de operații (tranzacțiile) se garantează fie că vor fi complet executate, fie că nu vor avea nici un efect. În oricare dintre situații, baza de date rămâne într-o stare consistentă.

Furnizori de baze de date relaționale

Pe piață există multe SGDB-uri. O să amintim doar câteva firme producătoare importante:

- **Oracle**, cu SGDB-ul ***Oracle***. Deține recordul de vânzări în acest moment.
- **IBM** cu produsele ***DB2*** și ***Informix***.
- **Microsoft** cu ***SQL Server 2005***.
- **Sybase**, cu SGDB-ul ***Sybase***.

Există și alte firme ale căror sisteme de gestiune a bazelor de date au un segment de piață bine delimitat, dar mult mai restrâns decât a celor menționate.

În acest punct trebuie să adăugăm rolul tot mai important pe care îl are în acest moment SGDB-ul **MySQL**. Se utilizează ca *free software*, sub *GNU General Public Licence* (GPL). Este foarte popular îndeosebi pentru aplicații Web, fiind disponibil pentru mai multe sisteme de operare.

Tehnologia ADO.NET – introducere

Aproape toate aplicațiile software interacționează cu baze de date. Este nevoie de un mecanism prin care aplicațiile se conectează și utilizează bazele de date. Pentru aplicațiile **.NET**, acest mecanism se numește **ADO.NET**.

Caracteristicile tehnologiei ADO.NET

ADO.NET este format dintr-un subset al ***Bibliotecii de Clase .NET***, folosite în programare pentru accesarea surselor de date, în special a bazelor de date relaționale.

Înainte ca platforma **.NET** să existe, în programarea aplicațiilor pentru sistemele *Windows* se utilizau următoarele tehnologii de accesare a bazelor de date: **ODBC** (*Open Database Connectivity*), ***OLE DB*** (*Object Linking and Embedding, Database*) și **ADO** (*ActiveX Data Objects*). **ADO** este o colecție de obiecte pentru accesarea surselor de date.

Pentru compatibilitate cu SGDB-uri mai vechi, **ADO.NET** include în continuare **ODBC** şi **OLE DB**, însă **ADO.NET nu este ADO**.

ADO.NET este o tehnologie complet nouă de acces la date. Este parte integrantă a platformei .NET şi nu este formată din obiecte *ActiveX*. Raţiunile pentru care Microsoft a păstrat cuvântul ADO în numele ADO.NET ţin de ideea că interfaţa de utilizare a celor două tehnologii este oarecum asemănătoare.

ADO.NET facilitează dezvoltarea de aplicaţii mai performante cu baze din următoarele considerente :

1. Accesul deconectat la bazele de date

ADO.NET este conceput să permită atât **accesul conectat** cât şi **deconectat** la bazele de date. Imaginaţi-vă că v-aţi conectat prin intermediul unei aplicaţii la un server de baze de date, aflat undeva la distanţă în reţea. Deschideţi o conexiune, citiţi datele, modificaţi datele şi la sfârşit vă deconectaţi. Acesta este ***modelul conectat***. Dezavantajul este că în tot acest timp comunicaţi prin reţea direct cu serverul care trebuie să menţină conexiunea şi să execute interogările. Dacă ne gândim că poate fi vorba de sute sau mii de utilizatori care transferă cantităţi mari de date, atunci înţelegem că în *modelul conectat*, utilizat cu vechile tehnologii, pot să survină supraîncărcari ale serverului şi ale traficului în reţea, un consum mare de resurse şi încetinirea sau blocarea aplicaţiilor.

În ***modelul deconectat*** datele sunt trimise de la server şi sunt stocate local la client într-o structură de date numită ***dataset***. Clientul operează doar asupra datelor din acest dataset, iar când a terminat, datele schimbate se trimit la server, acolo unde se găseşte adevărata bază de date. Astfel, serverul este eliberat de sarcina de a menţine conexiunile în mod permanent şi poate servi mai mulţi cu clienţi. Pe partea clientului de asemenea, toate operaţiile se desfăşoară mai rapid.

2. Integrarea XML.

ADO.NET este strâns legat de **XML**. XML este folosit intern de ADO.NET ca să menţină datele în ***dataset***-uri şi să reţină relaţiile şi constrângerile între tabele. Mai mult, suportul pentru XML este integrat in ADO.NET. Se pot produce *scheme* XML, se pot transmite date cu ajutorul *documentelor XML*.

Arhitectura ADO.NET

ADO.NET are două componente centrale:

1. ***Furnizorii de date*** (***data provider***)
2. ***Seturile de date (datasets)***

1. Furnizorii de date

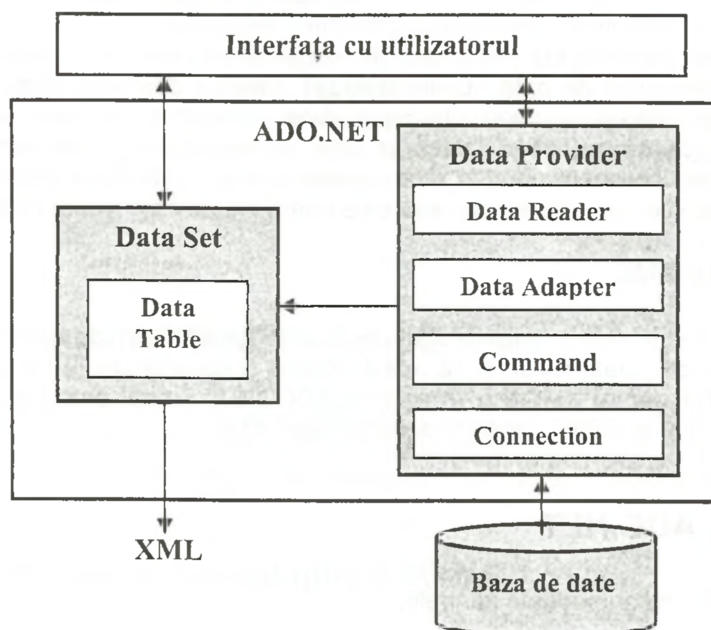
Un furnizor de date (*data provider)* asigură conectarea la o *sursă de date* (data source). Sursa de date poate fi un fişier text, un document XML, o bază de date, etc. Totodată un *furnizor de date* suportă accesul la date şi manipularea

datelor. Este legătura între aplicația dumneavoastră şi sursa de date. De regulă sursa de date este o bază de date şi atunci un *furnizor de date* asigură legătura între aplicație şi sistemul de gestiune a bazei de date respective.

Un *furnizor de date* se compune dintr-un număr de obiecte ale unor clase specializate. Aceste clase sunt definite în interiorul unor spații de nume speciale. Tabelul prezintă câteva dintre spațiile de nume în care sunt grupate componentele .NET:

Spațiul de nume	Descriere
`System.Data`	Clase, interfețe, delegări care definesc arhitectura **.NET**. Aici sunt clasele care definesc *dataset*-urile
`System.Data.Odbc`	*Data provider* .NET pentru **ODBC**
`System.Data.OleDb`	*Data provider* .NET pentru **OLE DB**
`System.Data.Sql`	Clase care suportă funcționalitate specifică SQL Server
`System.Data.OracleClient`	*Data provider* .NET pentru **Oracle**
`System.Data.SqlClient`	*Data provider* .NET pentru **SQL Server**

Figura 10.1. *Schema părților componente ale arhitecturii .NET.*



Sarcinile pe care le îndeplineşte un *data provider*:

- Furnizează accesul la date printr-o conexiune activă cu sursa de date.
- Asigură transmisia datelor la şi dinspre ***dataset***-uri (în modelul deconectat).
- Asigură transmisia datelor la şi dinspre aplicație (în modelul conectat)

Clasele unui *data provider* sunt:

`Connection`	Creează conexiunea cu sursa de date.
`Command`	Este utilizată pentru operaţii asupra sursei de date: citire, modificare, ştergere de date.
`Parameter`	Descrie un singur parametru al unei comenzi. De pildă, parametrul unei proceduri stocate.
`DataAdapter`	Este un adaptor pentru transferul datelor între sursa de date şi *dataset*.
`DataReader`	Este folosit pentru accesarea şi citirea rapidă a a datelor într-o sursă de date.

Pentru a înţelege cum sunt definite în .NET aceste clase, vom lista numele acestor clase pentru doi *furnizori de date*: ***Sql Server*** şi ***OLE DB .NET***. Reamintim că în spaţiul de nume **System.Data.SqlClient** sunt clasele *furnizorului de date **Sql Server***, iar în spaţiul de nume **System.Data.OleDb** sunt cele ale furnizorului de date, ***OLE DB***.
Clasele specifice celor doi *data provider*-i sunt:

*Furnizorul **OLE DB .NET***	*Furnizorul **SQL Server***	Corespunde clasei
`OleDbConnection`	`SqlConnection`	`Connection`
`OleDbCommand`	`SqlCommand`	`Command`
`OleDbDataReader`	`SqlDataReader`	`DataReader`
`OleDbDataAdapter`	`SqlDataAdapter`	`DataAdapter`

Pentru ca lucrurile să se lămurească pe deplin, enumerăm clasele **`DataReader`** specifice tuturor furnizorilor .NET:

- **`System.Data.SqlClient.SqlDataReader`**
- **`System.Data.OleDb.OleDbDataReader`**
- **`System.Data.Odbc.OdbcDataReader`**
- **`Oracle.OracleClient.OracleDataReader`**

2. *Seturile de date*

Dataset-ul este o componentă majoră a arhitecturii .NET. **`DataSet`** este clasa care defineşte un *dataset*. Un *dataset* este o colecţie de obiecte **`DataTable`** care pot fi legate între ele cu obiecte de tip **`DataRelation`**. *Dataset*-urile preiau de la *data provider*-i informaţiile necesare din sursa de date sub forma unei cópii reduse a bazei de date. Prin urmare, un *dataset* poate include o bază de date relaţională cu tabele, relaţii, vederi. Utilizatorul operează asupra tabelelor din *dataset*, care sunt practic o copie a tabelelor reale din baza de date. Deoarece un *dataset* este un obiect, *cahe*-ul pentru datele sale este în memorie.

Dataset-urile implementează *modelul de lucru deconectat* de baza de date. Când este necesar, *dataset*-urile transmit bazei de date modificările operate asupra datelor.

Clasa **`DataSet`**, ca şi celelalte clase care lucrează cu un *dataset* se găsesc în spaţiul de nume **System.Data**. Tabelul următor descrie aceste clase:

`DataSet`	Obiectele sale descriu schema întregii baze de date sau a unei submulțimi a sa. Conține tabele și relațiile între ele.
`DataTable`	Obiectele reprezintă o singură tabelă din baza de date. Conține rânduri și coloane.
`DataRow`	Reprezintă un singur rând într-o tabelă.
`DataColumn`	Reprezintă o coloană într-o tabelă.
`DataView`	Obiectele sortează datele. Nu sunt incluse într-un `DataTable`
`DataRowView`	Obiectele reprezintă un singur rând într-un `DataView`.
`DataRelation`	Reprezintă o relație între tabele. De exemplu, o relație de *cheie primară-cheie străină*
`Constraint`	Descrie o constrângere în tabelă, cum ar fi unicitatea valorilor într-o coloană cheie primară.

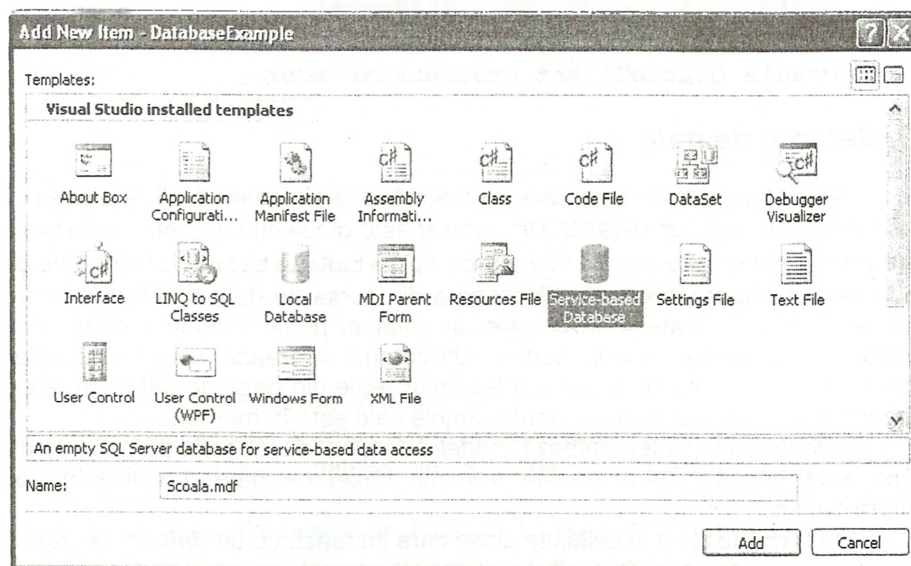
Crearea unei baze de date în VCSE

Construim o bază de date în cadrul unui proiect de tip consolă. Pentru aceasta, deschideți *Visual C# Express 2008* și urmați următoarele indicații:

1. Creați un nou proiect de tip consolă cu numele *DatabaseExample.*

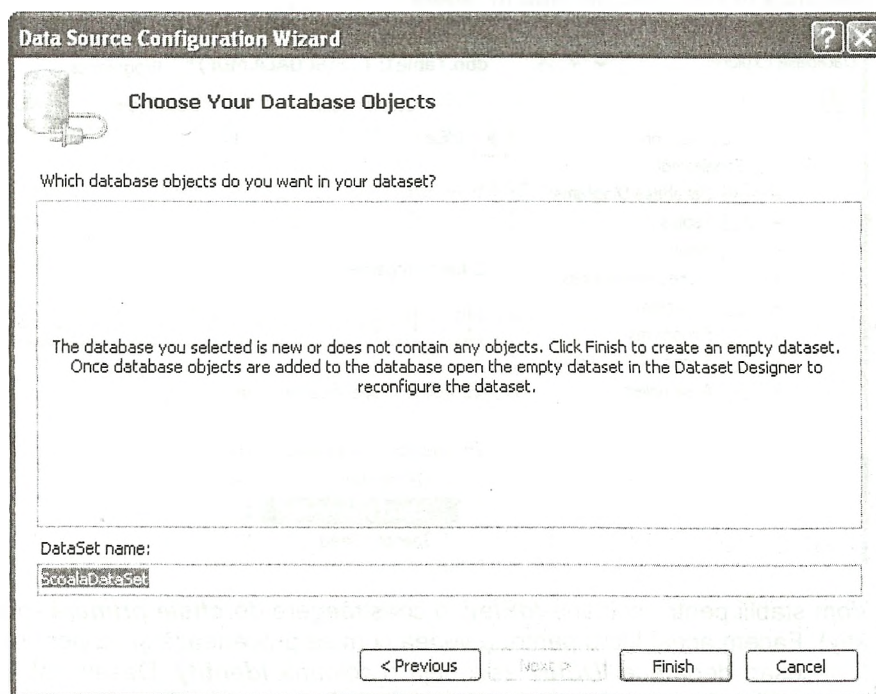
2. În meniul *View*, selectați *Other Windows*, apoi *Database Explorer*. Aceasta este o fereastră importantă, în care vizualizați și manipulați elementele bazei de date pe care o veți crea.

3. În *Solution Explorer*, acționați click drept pe numele proiectului, alegeți *Add*, apoi *New Item...* În dialogul *Add New Item*, selectați iconul *Service-based Database*, apoi numiți baza de date **`Scoala`**.



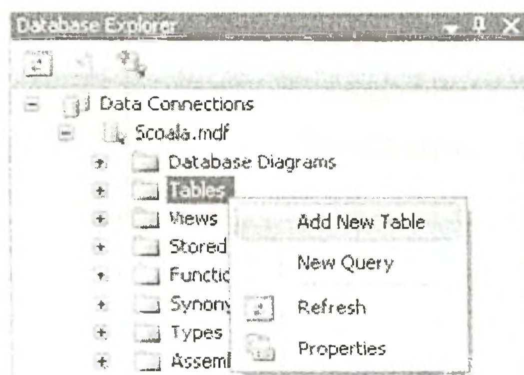
Remarcaţi că există şi alternativa de a alege *Local Database*. În această situaţie, baza de date este creată de către ***SQL Server Compact Edition***. Are extensia **.sdf**, şi este destinată să fie utilizată local. Testaţi şi această variantă.

4. În panoul *Add New Item*, acţionaţi butonul *Add*. Dialogul *Data Source Configuration Wizard* vă anunţă că va construi un *dataset* vid, pe care îl puteţi configura ulterior. Acceptaţi numele `ScoalaDataSet` pentru acesta:

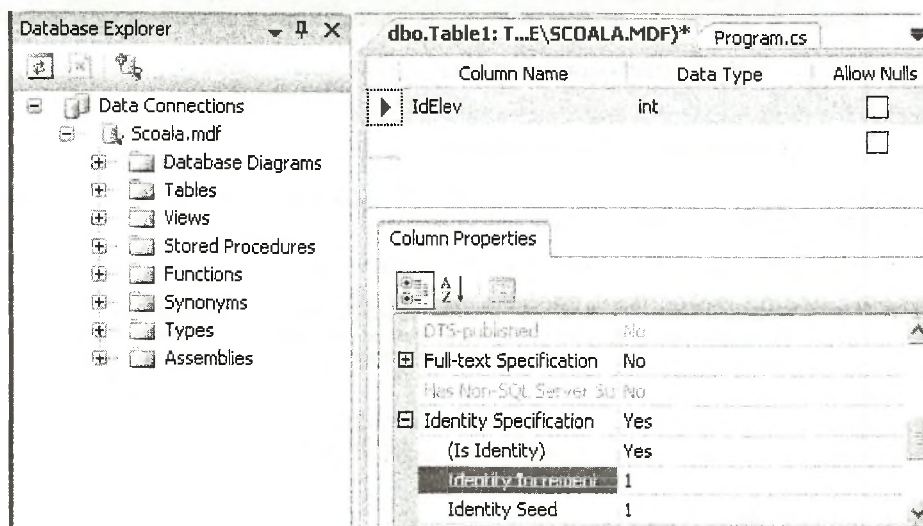


Apăsaţi butonul *Finish.*

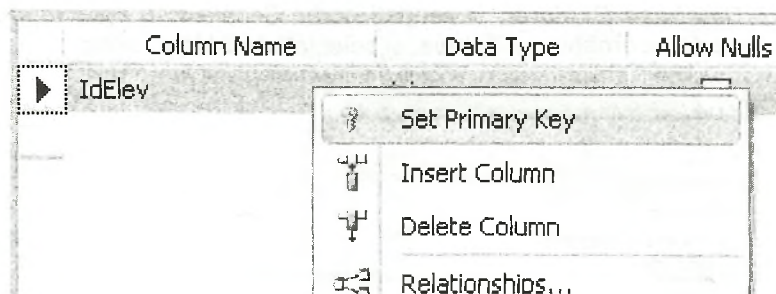
5. În panoul ***Database Explorer***, expandaţi nodul *Scoala.mdf*, care reprezintă baza de date. Click dreapta pe **Tables**, şi selectaţi *Add New Table*:



6. Panoul care se deschide este ***Table Designer***. Acesta vă permite să definiţi schema tabelei, adică numele, tipul şi constrângerile pentru fiecare coloană. Completaţi prima linie ca în figură. În fereastra *Column Properties*, setaţi pentru câmpul ***IdElev***, proprietatea `IsIdentity` la valoarea *Yes* şi *Identity Increment* la valoarea **1**. Aceasta înseamnă că tipul coloanei trebuie să fie **int**, iar valorile pe coloană se incrementează în mod automat începând de la valoarea **1**, pentru fiecare rând nou introdus. În acest fel, ***IdElev*** poate identifica în mod unic un rând în tabelă:



7. Vom stabili pentru coloana ***IdElev***, o constrângere de ***cheie primară*** (*primary key*). Facem acest lucru pentru a vedea cum se procedează şi nu pentru că ar fi necesar, deoarece ***IDElev*** este deja o coloană ***Identity***. Deselectaţi *check box*-ul *Allow Nulls*, deoarece o coloană *Identity* sau o cheie primară nu admite valori nule. Selectaţi în *Table Designer* primul rând, faceţi click dreapta şi alegeţi *Set Primary Key*:

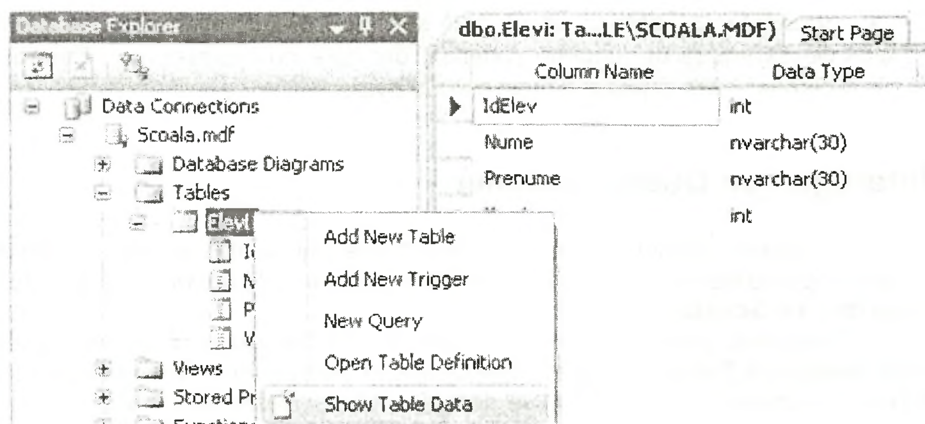


8. Definiţi celelalte coloane astfel:

	Column Name	Data Type	Allow Nulls
	IdElev	int	☐
	Nume	nvarchar(30)	☐
	Prenume	nvarchar(30)	☑
▶	Varsta	int (int, money, nchar(10))	☑
			☐

9. Salvaţi proiectul şi tabela apăsând butonul *Save All*. În dialogul *Choose Name* introduceţi numele tabelei: **Elevi**.

10. Pentru a însera (*design time*) câteva rânduri cu date în tabelă, expandaţi nodul *Tables* în ***Database Explorer***. Click dreapta pe tabela **`Elevi`** şi selectaţi *Show Table Data*.



11. S-a deschis *Query Designer*. Aici puteţi introduce manual date în tabelă şi puteţi efectua interogări. Completaţi câteva rînduri, ca mai jos. Observaţi că pe coloana **`IdElev`** serverul nu vă permite să introduceţi valori, deoarece este coloană ***Identity*** şi aceasta înseamnă că **SQL Server** ii atribuie valori în mod automat:

Elevi: Query(...LE\SCOALA.MDF) | dbo.Elevi: Ta...LE\SCOALA.MDF) | St

	IdElev	Nume	Prenume	Varsta
▶	1	Pascu	Ioan	18
	2	Nedelcu	Valentin	17
	3	Pop	Andrei	19
	4	Ilie	Ioan	18
*	NULL	NULL	NULL	NULL

12. Dacă doriți să adăugați și alte tabele bazei de date ***Scoala***, de pildă tabelele `Clase`, `Profesori`, `Materii`, e foarte simplu: faceți click dreapta pe **Tables** în ***Database Explorer*** și alegeți *New Table.*, apoi repetați pașii **6...11**.

Nu veți face ***build*** acestui proiect, pentru că aplicația nu execută nimic. Vom vedea în secțiunile următoare cum ne conectăm la baza de date și cum o interogăm programatic.

De reținut:

- O **cheie primară** (***primary key***) identifică în mod unic fiecare rând într-o tabelă. Cheia primară include o coloană sau o combinație de coloane. Nu pot exista două rânduri distincte într-o tabelă care să aibă aceeași valoare (sau combinație de valori) în aceste coloane.
- O **coloană identitate** (***Identity column***) identifică în mod unic fiecare rând într-o tabelă, iar valorile sale sunt generate de către baza de date. Deseori o coloană identitate e desemnată și cheie primară.
- **`varchar(n)`** este un tip de dată specifică SGDB-urilor, care reține un set de caractere de lungime variabilă, dar care nu depășește `n`. Pentru *SQL Server 2005*, `n` poate fi maxim 8000 *bytes*.

Interogări cu Query Designer

În această secțiune vom face câteva exerciții de manipulare a datelor, în scopul împrospătării cunoștințelor referitoare la comenzile limbajului **SQL**. Utilizăm baza de date ***Scoala***, construită anterior.

Deschideți proiectul *DatabaseExample*. În ***Database Explorer***, acționați click dreapta pe **Tables** și alegeți *New Query*. În dialogul *Add Table* selectați tabela `Elevi` și acționați butonul *Add*. Se deschide *Query Designer*-ul mediului integrat, care vizual consistă din patru panouri: *Panoul Diagramelor* (*Diagram Pane*), *Panoul Criteriilor* (*Criteria Pane*), *Panoul SQL* (*SQL Pane*), *Panoul de Vizualizare a Rezulatelor* (*Show Results Pane*).

În *Panoul SQL* vom introduce interogări, iar rezultatele interogării, adică setul de rezultate, este vizualizat în *Panoul Rezultatelor*. Panoul diagramelor permite selectarea coloanelor iar cel al criteriilor servește la stabilirea criteriilor de sortare. Toate selecțiile făcute în cele două panouri generează cod în *Panoul SQL*.

Reamintim faptul că limbajul **SQL** *(Structured Query Language)* include două secțiuni importante:

1. **Data Manipulation Language** (**DML**).
   DML constă din interogări și comenzi pentru modificarea datelor. Acestea sunt: `SELECT`, `UPDATE`, `DELETE` și `INSERT INTO`.

2. **Data Definition Language** (**DDL**).
   Comenzile **DDL** permit crearea și ștergerea tabelelor într-o bază de date. Cele mai importante comenzi de acest tip sunt: `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`.

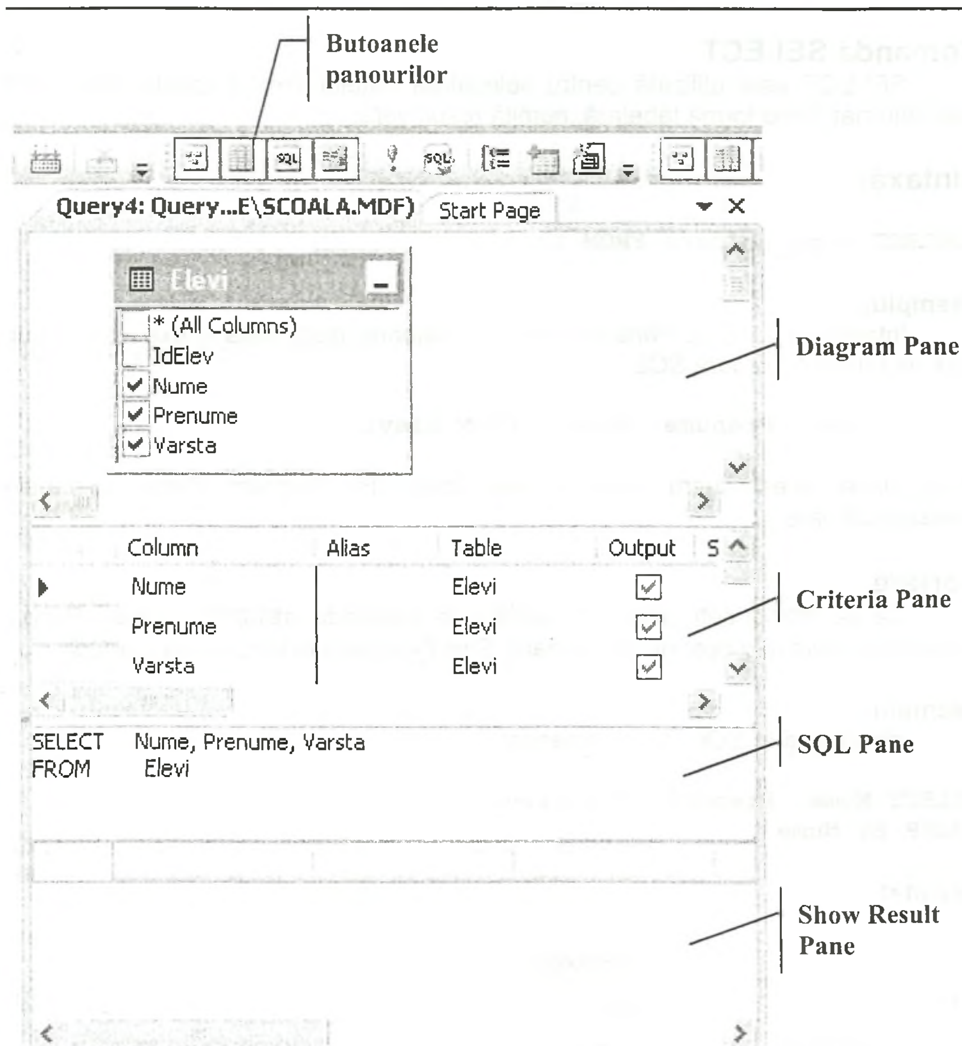


Figura 10.2 *Query Designer*

Executarea interogării

Pentru executare acţionaţi click pe butonul *Execute SQL* (butonul cu semnul exclamării) sau de la tastatură **Ctrl + R**.
În panoul de rezultate, se afişează:

	Nume	Prenume	Varsta
▶	Pascu	Ioan	18
	Nedelcu	Valentin	17
	Pop	Andrei	19
	Ilie	Ioan	18

Comanda SELECT

SELECT este utilizată pentru selecatrea datelor dintr-o tabelă. Rezultatul este returnat într-o formă tabelară, numită *result set*.

Sintaxă:

```
SELECT nume_coloane FROM tabela
```

Exemplu:

Introduceți în *SQL Pane* comanda următoare, după care o executați cu un click pe butonul *Execute SQL*:

```
SELECT Nume, Prenume, Varsta FROM Elevi
```

Puteți face acest lucru manual, sau bifați în *Diagram Pane* coloanele corespunzătoare.

Sortare

Ca să introduceți criterii de sortare în comanda **SELECT**, utilizați *Panoul Criteriilor* (selectați o opțiune din coloana *Sort Type*) sau le introduceți manual.

Exemplu:

Introduceți în *SQL Pane* comanda:

```
SELECT Nume, Prenume FROM Elevi
ORDER BY Nume
```

Rezultat:

	Nume	Prenume
▶	Ilie	Ioan
	Nedelcu	Valentin
	Pascu	Ioan
	Pop	Andrei

Clauza WHERE

WHERE permite selecția condiționată. Cu * selectați toate coloanele tabelei.

Exemplu:

Introduceți în *SQL Pane* comanda:

```
SELECT * FROM Elevi
WHERE Varsta >= 18
```

Rezultat:

	IdElev	Nume	Prenume	Varsta
▶	1	Pascu	Ioan	18
	3	Pop	Andrei	19
	4	Ilie	Ioan	18

Comanda INSERT INTO

Comanda inserează un rând nou în tabelă.

Sintaxă:

```
INSERT INTO nume_tabelă
VALUES valoare1, valoare2, ...
```

Puteţi de asemenea să specificaţi coloanele în care se inserează date:

```
INSERT INTO nume_tabelă (coloana1, coloana2, ...)
VALUES valoare1, valoare2, ...
```

Exemplu:

Introduceţi în *SQL Pane*:

```
INSERT INTO Elevi (Nume, Prenume, Varsta)
VALUES ('Florea', 'Constantin', 99)
```

Rezultat:

Pentru a vedea rezultatul, avem nevoie de un *result set* întors de o instrucţiune **SELECT**. Introduceţi interogarea:

```
SELECT * FROM Elevi
```

	IdElev	Nume	Prenume	Varsta
▶	1	Pascu	Ioan	18
	2	Nedelcu	Valentin	17
	3	Pop	Andrei	19
	4	Ilie	Ioan	18
	7	Florea	Constantin	99

Comanda UPDATE

Comanda este utilizată pentru modificarea datelor într-o tabelă.

Sintaxă:

```
UPDATE nume_tabelă
SET coloană1 = valoare1, coloană2 = valoare2, ...
WHERE coloană = valoare
```

Cu **UPDATE** puteți modifica valorile pe una sau mai multe coloane, în rândurile selectate cu clauza **WHERE**.

Exemplu:

Introduceți în *SQL Pane*:

```
UPDATE Elevi
SET Varsta = 104
WHERE Prenume = 'Ioan'
```

Rezultat:

Pentru a vizualiza rezultatul introduceți în *SQL Pane* interogarea:

```
SELECT * FROM Elevi
```

	IdElev	Nume	Prenume	Varsta
▶	1	Pascu	Ioan	104
	2	Nedelcu	Valentin	17
	3	Pop	Andrei	19
	4	Ilie	Ioan	104
	7	Florea	Constantin	99

Comanda DELETE

Comanda este utilizată pentru ştergerea rândurilor dintr-o tabelă.

Sintaxă:

```
DELETE FROM nume_tabelă
WHERE coloană = valoare
```

Exemplu:

Introduceți în *SQL Pane* comanda următoare, după care executați-o cu click pe butonul *Execute SQL*:

```
DELETE FROM Elevi
WHERE Varsta > 19
```

Rezultat:

Pentru a vizualiza rezultatul introduceți în *SQL Pane* interogarea:

```
SELECT * FROM Elevi
```

	IdElev	Nume	Prenume	Varsta
▶	2	Nedelcu	Valentin	17
	3	Pop	Andrei	19

Comanda CREATE TABLE

Comanda creează o tabelă într-o bază de date.

Sintaxă:

```
CREATE TABLE nume_tabelă
(
coloana1 tip_dată,
coloana2 tip_dată,
...
)
```

Exemplu:

Lucrăm cu aceeaşi bază de date, **Scoala**. Introduceţi în *SQL Pane* comanda următoare, după care executaţi cu un click pe butonul *Execute SQL*:

```
CREATE TABLE Clase (IDClasa varchar(5), Profil varchar(20),
                    NrElevi int)
```

Rezultat:

În ***Database Explorer*** a apărut tabela **Clase**. Acţionaţi click drept asupra numelui clasei şi selectaţi *Open Table Definition*:

	Column Name	Data Type	Allow Nulls
▶	IDClasa	varchar(5)	☑
	Profil	varchar(20)	☑
	NrElevi	int	☑

Comanda ALTER TABLE

Comanda este utilizată pentru a adăuga sau a şterge coloane într-o tabelă.

Sintaxă:

```
ALTER TABLE tabelă
ADD coloană tip

ALTER TABLE tabelă
DROP COLUMN coloană
```

Exemplu:

Introduceți în *SQL Pane* comanda următoare, după care o executați cu un click pe butonul *Execute SQL*:

```
ALTER TABLE Clase
ADD Diriginte varchar(20)
```

Rezultat:

În ***Database Explorer*** acționați click drept asupra numelui clasei și selectați *Open Table Definition*:

Column Name	Data Type	Allow Nulls
IDClasa	varchar(5)	☑
Profil	varchar(20)	☑
NrElevi	int	☑
Diriginte	varchar(20)	☑

Exemplu:

Introduceți în *SQL Pane* comanda pentru ștergerea coloanei **`Diriginte`**:

```
ALTER TABLE Clase
DROP COLUMN Diriginte
```

Rezultat:

Column Name	Data Type	Allow Nulls
IDClasa	varchar(5)	☑
Profil	varchar(20)	☑
NrElevi	int	☑

Comanda DROP TABLE

Comanda este utilizată pentru a șterge o tabelă într-o bază de date.

Sintaxă:

```
DROP TABLE tabelă
```

Exemplu:

Vom șterge tabela **`Clase`**. Introduceți în *SQL Pane* comanda următoare, după care o executați cu un click pe butonul *Execute SQL*:

```
DROP TABLE Clase
```

Rezultat:

În *Database Explorer*, click dreapta pe **Tables** şi selectaţi *Refresh*. Se observă că tabela `Clase` a fost ştearsă.

Controlul DataGridView

Acest control este vedeta controalelor pentru afişarea datelor. Clasa `DataGridView` permite construirea unei tabele adaptabile. Celulele, rândurile, coloanele se setează prin intermediul proprietăţilor `Rows` şi `Columns`.

Controlul se poate popula manual sau programatic. Ca alternativă, controlul se poate "*lega*" la o sursă de date prin intermediul proprietăţilor `DataSource` şi `DataMember`. Astfel popularea se face în mod automat.
Prezentăm un exemplu de manipulare simplă a acestui control.

Aplicaţia DataGridViewExample

Realizăm un proiect de tip consolă, care conţine o bază de date cu o singură tabelă. Informaţiile din tabelă se vor afişa într-un control **DataGridView** în momentul pornirii aplicaţiei. Urmaţi paşii:

1. Creaţi un proiect de tip *Windows Forms* cu numele *DataGridViewExample*.

2. Construiţi o bază de date cu numele **Sport**. Adăugaţi în baza de date tabela `Sportivi`. Tabela are următoarea structură:

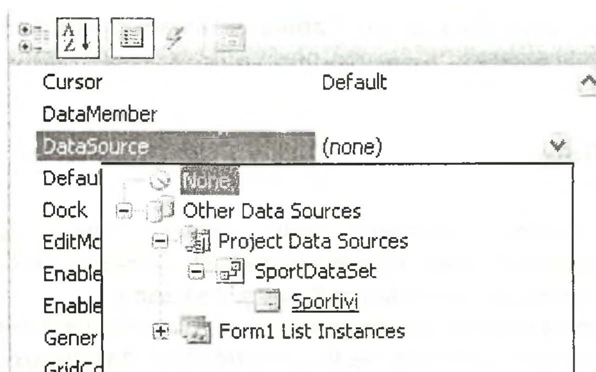
Column Name	Data Type	Allow Nulls
IdSportiv	int	☐
Nume	nvarchar(15)	☐
Prenume	nvarchar(15)	☑
DataNasterii	datetime	☑
SportulPracticat	nvarchar(30)	☐

Pentru construirea bazei de date şi a tabelei, urmaţi indicaţiile temei: "**Crearea unei baze de date în VCSE**".

3. Introduceţi câteva înregistrări în tabelă. Pentru aceasta, selectaţi tabela `Sportivi` în ***Database Explorer***, acţionaţi click dreapta şi alegeţi *Show Table Data*.

4. Din **Toolbox**, trageţi pe suprafaţa formei un control de tip **DataGridView** şi setaţi proprietatea `Dock` la valoarea `Fill`.

5. Selectaţi controlul. În fereastra **Properties**, atribuiţi proprietăţii `DataSource` valoarea `Sportivi`. Se face acest lucru, expandând în panoul din dreapta în mod succesiv nodurile: *Other Data Sources*, *Project Data Sources* şi *SportDataSet*:



Astfel, în mod vizual, aţi legat controlul la sursa de date.

6. Faceţi ***Build*** şi rulaţi cu **F5**.

La rulare obţineţi:

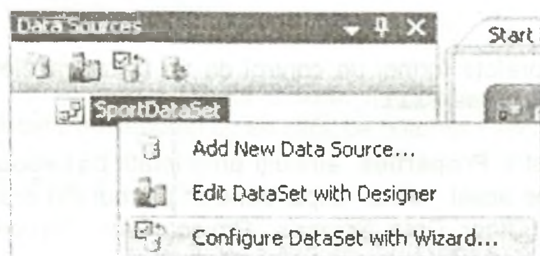
Form1

IdSportiv	Nume	Prenume	DataNasterii	SportulPracticat
1	Dita Tomescu	Constantina	23.01.1970	Atletism
2	Izbasa	Sandra	28.05.1998	Gimnastica
3	Dumitru	Alina	30.08.1982	Judo
4	Andrunache	Georgeta	14.04.1976	Canotaj
5	Susanu	Viorica	29.10.1975	Canotaj

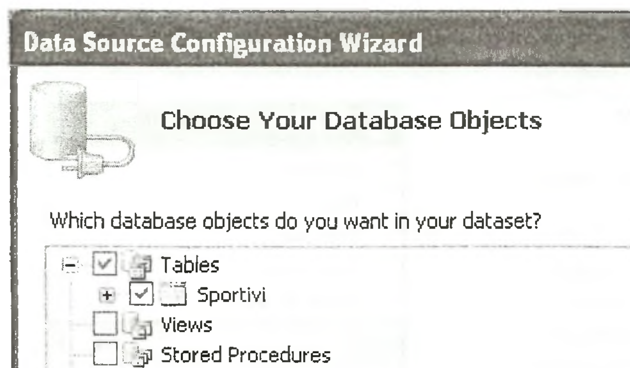
Există şi alte variante vizuale pentru popularea controlului.

Varianta a 2-a
Procedaţi după cum urmează:
Paşii **1**, **2**, **3**, **4** sunt identici cu cei specificaţi anterior. Continuaţi astfel:

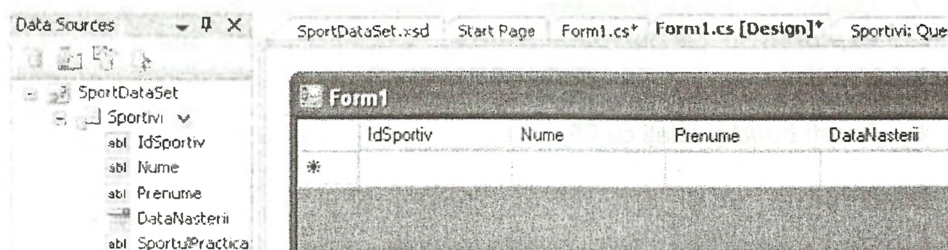
5. În meniul ***Data***, selectaţi *Show Data Sources*. În fereastra ***Data Sources***, acţionaţi click dreapta pe elementul **`SportDataSet`**, apoi selectaţi *Configure DataSet with Wizard...*



6. În dialogul *Data Source Configuration Wizard*, selectaţi tabela **`Sportivi`**, apoi apăsaţi butonul *Finish*:



7. În fereastra ***Data Source***, selectaţi tabela **`Sportivi`** şi trageţi-o pur şi simplu cu mouse-ul deasupra controlului **DataGridView**, în *Form Designer*.



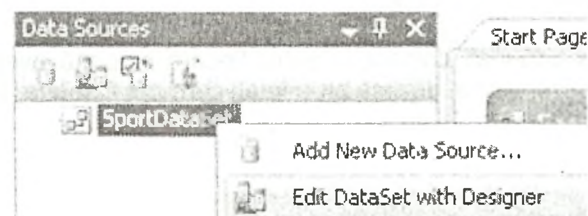
8. Faceţi ***Build*** şi rulaţi cu **F5**. Veţi obţine popularea controlului cu datele din tabelă.

Varianta a 3-a

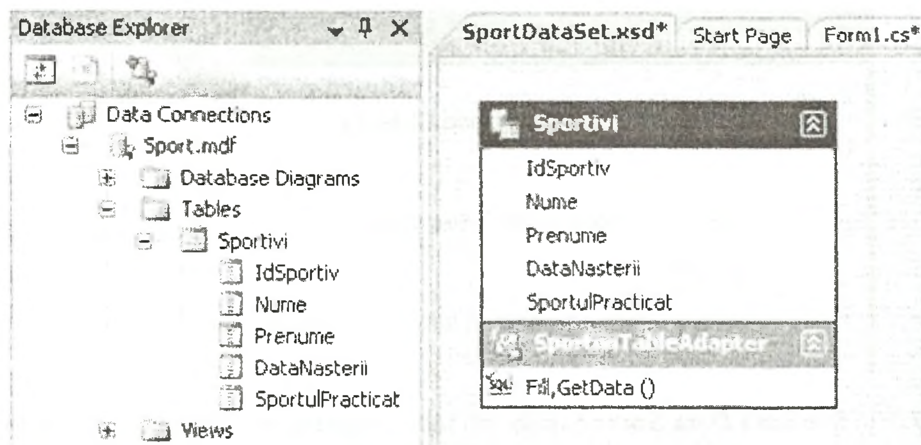
Procedaţi astfel:

Paşii 1, 2, 3, 4 sunt identici cu cei ai aplicaţiei anterioare. Continuaţi astfel:

5. Deschideţi ***DataSet Designer***. Pentru aceasta, aveţi două posibilităţi: în ***Solution Explorer*** faceţi dublu click pe itemul **`SportDataSet.xsd`** sau în panoul ***Data Source***, click dreapta pe itemul **`SportDataSet`** şi selectaţi *Edit DataSet with Designer*.



6. În ***Database Explorer***, selectaţi tabela **`Sportivi`** şi o trageţi cu mouse-ul pe suprafaţa ***DataSet Designer***-ului:



7. Deschideţi ***Form Designer***-ul cu dublu click pe **`Form1.cs`** în ***Solution Explorer***. În panoul ***Data Source***, selectaţi cu click tabela **`Sportivi`** şi plasaţi-o pe suprafaţa controlului **DataGridView**.

8. Faceţi ***build*** şi rulaţi cu **F5**.

Observaţii:

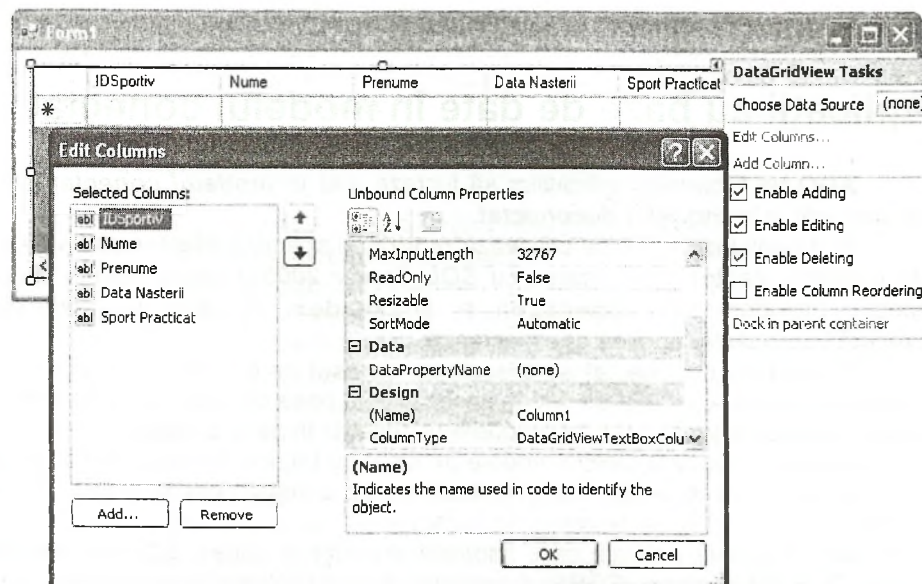
- Toate cele trei variante de populare a controlului **DataGridView** realizează o ***legare*** a controlului la sursa de date printr-un obiect de tip **`BindingSource`**.
- Despre clasele **`DataSet`**, **`BindingSource`**, **`Table Adapter`**, dar şi despre modul în care populaţi programatic un control **DataGridView**, vom discuta în secţiunile următoare.

Configurarea *design time* a coloanelor unui DataGridView

Coloanele se adaugă şi se configurează astfel:

1. În *Form Desiger* acţionaţi click pe săgeata din colţul dreapta sus al controlului **DataGridView**, apoi alegeţi *Edit Columns*.

2. În dialogul *Edit Columns* adăugaţi coloane. Pentru fiecare coloană aveţi în panoul *Unbound Column Properties*, diverse proprietăţi care pot fi setate, între care şi textele *header*-elor.


Form1
IDSportiv
Nume
Prenume
Data Nasterii
Sport Practicat
DataGridView Tasks
Choose Data Source
(none)
Edit Columns...
Add Column...
Enable Adding
Enable Editing
Enable Deleting
Enable Column Reordering
Dock in parent container
Edit Columns
Selected Columns:
IDSportiv
Nume
Prenume
Data Nasterii
Sport Practicat
Add...
Remove
Unbound Column Properties
MaxInputLength 32767
ReadOnly False
Resizable True
SortMode Automatic
Data
DataPropertyName (none)
Design
(Name) Column1
ColumnType DataGridViewTextBoxColu
(Name)
Indicates the name used in code to identify the object.
OK
Cancel

Capitolul 11

Aplicații cu baze de date în modelul conectat

ADO.NET permite aplicațiilor să lucreze atât în ***modelul conectat*** la baza de date, cât şi în ***modelul deconectat***.

În modelul conectat se utilizează provider-ii specifici diferitor surse de date. De exemplu, pentru comunicarea cu ***SQL Server 2005***, folosim clasele specifice acestui provider: **`SqlConnection`** şi **`SqlReader`**. Aplicația se conectează, execută interogările, apoi se deconectează.

În modelul deconectat se crează un ***dataset*** care preia din baza de date tablele necesare. Aplicația se deconectează de la baza de date, execută interogări asupra acestui *dataset*, apoi face *update*-ul necesar în baza de date.

În continuare propunem câteva modele de lucru cu bazele de date, atât în modelul conectat cât şi deconectat. Vor fi aplicații de tip consolă, dar şi de tip *Windows Forms*.

Vom realiza câteva aplicații care folosesc *provider*-ul pentru ***SQL Server 2005***, pentru **OLE DB** şi pentru **ODBC**. Reamintiți-vă că **ADO.NET** include *data provideri* pentru diferite surse de date.

Utilizarea provider-ului pentru SQL Server 2005

Realizăm o aplicație este de tip consolă. Reluăm aplicația *DatabaseExample* din capitolul anterior. Aceasta conține baza de date **`Scoala`**, cu o singură tabelă: **`Elevi`**. Să presupunem că tabela are doar două rânduri:

	IdElev	Nume	Prenume	Varsta
	1	Nedelcu	Valentin	17
	2	Pop	Andrei	19

Executăm două operații asupra tabelei **`Elev`**: inserarea unui rând nou şi selectarea rândurilor.

Urmați paşii:

1. Deschideți aplicația de tip consolă, *DatabaseExample*. În ***Solution Explorer***, redenumiți **`Program.cs`** astfel: **`SQLDataProvider.cs`** (click dreapta şi *Rename*).

2. Deschideți fişierul **`SQLDataProvider.cs`** (click dreapta şi *View Code*) apoi înlocuiți codul existent cu următorul:

```
using System;
using System.Data.SqlClient;
```

```
namespace ConsoleApplication1
{
   class SQLDataProvider
   {
       static void Main(string[] args)
       {
          // Construim stringul de conectare
          // (connection string)
          string connString =
                 @"server = .\sqlexpress;
                 Database=Scoala.mdf;
                 trusted_connection=True;
                 AttachDbFileName =
 C:\DatabaseExample\DatabaseExample\bin\Debug\Scoala.mdf";

          // Pregătim interogările
          string sql1 = @"SELECT * FROM Elevi";
          string sql2 = @"INSERT INTO Elevi
                         (Nume, Prenume, Varsta)
                         VALUES
                         ('Axente', 'Mihai', 20)";

          // Declarăm variabilele conexiune
          // şi cititorul de date
          SqlConnection conn = null;
          SqlDataReader reader = null;
          try
          {
              // Deschidem conexiunea
              conn = new SqlConnection(connString);
              conn.Open();

              // Executăm prima interogare
              SqlCommand cmd = new SqlCommand(sql1, conn);
              reader = cmd.ExecuteReader();

              // Închidem reader-ul
              // pentru a putea fi refolosit
              reader.Close();

              // Executăm a doua interogare - inserarea
              // unui rand
              cmd = new SqlCommand(sql2, conn);
              reader = cmd.ExecuteReader();
              reader.Close();

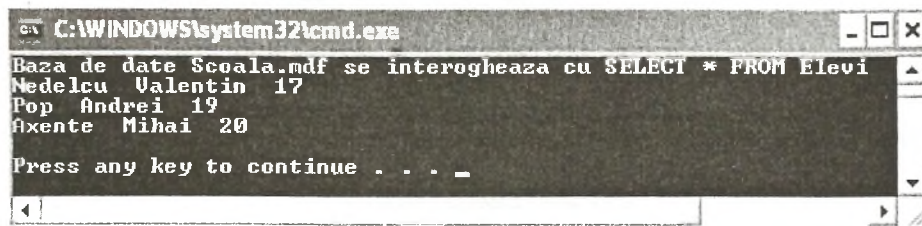
              // Din nou un SELECT pentru a avea
              // un result set
              cmd = new SqlCommand(sql1, conn);
```

```
                reader = cmd.ExecuteReader();

                Console.WriteLine("Baza de date " +
                   conn.Database + " se interogheaza cu "
                 + cmd.CommandText);

                // Afişăm informaţiile din result set
                while (reader.Read())
                   Console.WriteLine("{0}  {1}  {2}",
                       reader[1], reader["Prenume"],
                       reader[3]);
                Console.WriteLine();
            }
            catch (Exception e)
            {
                Console.WriteLine("Eroare: " + e);
            }
            finally
            {
                // Conexiunea se închide indiferent dacă se
                // aruncă sau nu excepţii
                reader.Close();
                conn.Close();
                conn.Dispose();
            }
        } //Main()
    }
}
```

3. Faceţi ***Build*** şi rulaţi cu **Ctrl + F5** (Modul depanare)



Reţineţi etapele comunicării cu baza de date:

Paşii următori sunt aceeaşi indiferent de providerul cu care lucraţi:

- **Se include spaţiul de nume `System.Data.SqlClient`.** Clasele acestuia definesc providerul pentru ***MS SQL Server***. Pentru alţi provideri, se înlocuieşte prefixul `Sql` cu cel corespunzător.

- **Se deschide o conexiune cu baza de date**

- Clasa `SqlConnection` permite obținerea unei conexiuni la server şi deschiderea conexiunii cu metoda `Open()`.
- Constructorul acestei clase, `SqlConnection(connString)`, primeşte stringul de conectare.
- Proprietățile `Database` şi `CommandText` ale clasei `SqlConnection` returnează baza de date, respectiv interogarea curentă.

Executarea interogării

- Ca să executați o interogare, trebuie mai întâi să construiți un obiect de tip `SqlCommand`.
- Constructorul `SqlCommand(sql,conn)` primeşte ca argumente interogarea care trebuie executată şi conexiunea prin care va lucra.
- Executararea propriuzisă a interogării, se face cu metoda `ExecuteReader()`. Aceasta face două lucruri : construieşte un obiect (`reader`) de tip `SqlDataReader` şi apoi execută interogarea.
- Metoda `ExecuteReader()` returnează un *result set* care poate fi vid pentru comenzi *non-query*, cum este `INSERT INTO`, dar nevid pentru `SELECT`. Acest *result set* se poate imagina ca o tabelă în care se regăsesc rezultatele interogării.

Accesarea rezultatelor interogării

- Ca să analizați setul de rezultate, parcurgeți setul în buclă cu metoda `Read()`, care la fiecare apel, avansează `reader`-ul la următoarea linie.
- Datele de pe linie se accesează indexat. `reader[0]` este valoarea corespunzătoare primei coloane, `reader[1]` corespunde celei de a doua coloane, etc. Un alt mod de accesare este prin numele coloanei respective, aşa cum am arătat în aplicație: `reader["Prenume"]` este valoarea aflată pe linia curentă şi coloana `Prenume`. Se preferă accesarea prin indecşi pentru că este mai rapidă.

Protecția împotriva excepțiilor

- Deschiderea conexiunii (`new SqlConnection(connString)`) trebuie protejată într-un bloc `try` ...`catch`. pentru tratarea excepțiilor. Astfel ne asigurăm că blocul `finally`, în care eliberăm resursele va fi executat. Se face acest lucru, deoarece **ADO.NET** aruncă excepții la erorile de baze de date. Este foarte simplu să aveți erori. De exemplu, greşiți ceva în stringul de conectare, sau baza de date este blocată de căte o altă aplicație.

Blocul `using` – o tehnică sigură

Clasele **Connection**, aşa cum este **SqlConnection** moştenesc interfaţa **IDisposable**. Aceasta înseamnă că un obiect de acest tip poate referi resurse negestionate în mod automat de **.NET** (*unmanaged resources*). Poate fi vorba de surse de date, fişiere pe disc şi altele. Toate clasele care moştenesc această interfaţă au metoda **Dispose()**, care trebuie invocată de către programator pentru eliberarea acestor resurse. În mod obişnuit, se apelează în blocul **finally**.

O metodă mai elegantă pentru tratarea excepţiilor este utilizarea *blocului* **using**. Nu confundaţi cu *directiva* **using**. Obiectele **IDispozable** se declară şi se instanţiază în declaraţia **using**. În acest fel, vă asiguraţi că metoda **Dispose()** este apelată chiar în cazul în care se aruncă excepţii.

Iată cum se poate scrie codul cuprins în blocul **try** ... **catch** în programul de mai sus:

```
using (conn = new SqlConnection(connString) )
{
    conn.Open();

    SqlCommand cmd = new SqlCommand(sql1, conn);
    reader = cmd.ExecuteReader();

    // cod ...

    while (reader.Read())
        Console.WriteLine("{0}  {1}  {2}",
             reader[1], reader["Prenume"],
             reader[3]);
    reader.Close();
}
```

Observaţi că nu mai este nevoie nici măcar să închideţi conexiunea, deoarece de acest lucru se ocupă acum C#.

Ce reprezintă *connection string* ?

Stringul de conectare este un string care specifică informaţii despre o sursă de date, precum şi mijloacele de conectare la aceasta. Se transmite *data provider*-ului pentru iniţierea conexiunii. Conţine atribute cum sunt : numele driverului, numele serverului de baze de date, numele bazei de date, informaţii de securitate cum ar fi user şi parolă şi altele.

În aplicaţia anterioară, **server = .\sqlexpress;** specifică faptul că se utilizează o instanţă a *SQL Server 2005 Express Edition*, aflat pe maşina locală (prin **.** se precizează că e vorba de *localhost*).
Se indică deasemenea numele bazei de date prin **Database=Scoala.mdf;** Mai departe, **trusted_connection=True;** permite userului curent autentificat să facă o conexiune.

Atributul **`AttachDbFileName`** specifică calea de director pe discul local până la baza de date. Procesul prin care aduceți la cunoștință serverului ***SQL Server*** de existența unei baze de date se numește ***atașarea*** bazei de date.

Ajunși în acest punct, precizăm că baza de date se compune din două fișiere strâns legate : fișierul cu extensia (.mdf) și cel cu extensia (.ldf). Primul depozitează datele, iar al doilea este fișier de ***log***, care conține informații despre tranzacții.

Există mai multe atribute care pot fi incluse în stringul de conectare. De multe ori, alegerea lor corectă este o problemă. Din fericire, există destulă documentație și chiar site-uri dedicate. Recomandăm unul dintre acestea: **www.connectionstrings.com**.

Utilizarea provider-ului pentru OLE DB

Cu ajutorul furnizorului de date **OLE DB**, se pot accesa date depozitate în diverse tipuri de baze de date, de la **MS Access** până la **Oracle**. Vă puteți conecta inclusiv la **SQL Server 2005**, însă acesta are propriul lui provider, așa cum ați văzut, care este mai rapid. Acest data provider este definit în spațiul de nume **System.Data.OleDb**. Cele mai importante clase ale sale sunt:

`OleDbCommand`	Execută interogări sau proceduri stocate
`OleDbConnection`	Reprezintă o conexiune la o sursă de date
`OleDbDataReader`	Permite citirea numai înainte, a unui stream de rânduri dintr-o sursă de date
`OleDbDataAdaptor`	Reprezintă o legătură între sursa de date și *dataset*, folosită pentru obținerea și salvarea datelor.

Este ușor de remarcat similitudinea între clasele furnizorului **OLE DB** și cele ale ***SQL Server 2005***. Metodele acestor clase sunt de asemenea similare.

Aplicația OleDbProviderWinApp

Vom accesa o bază de date *Microsoft Access 2003* cu ajutorul providerului **`Microsoft.Jet.OLEDB.4.0`**. Acest furnizor este specific pentru bazele de date **Access**.
Urmați procedura:

1. Deschideți *Micosoft Access* și creați baza de date *Access* **Persoane.mdb**, cu următoarea schemă:

Microsoft Access

Fişier Editare Vizualizare Inserare Instrumente

Persoane : Tabel

Nume câmp	Tip de date
ID	AutoNumerotare
Nume	Text
Prenume	Text
DataNasterii	Dată/Oră

Completaţi câteva rânduri cu date în tabelă. Salvaţi baza de date **Persoane.mdb** în folderul **C:\teste**.

2. Creaţi în **VCSE** un nou proiect de tip *Windows Forms*, cu numele *OleDbProviderWinApp*.

3. Pe suprafaţa formei plasaţi un control de tip **DataGridView**.

4. În fişierul *Form1.cs* introduceţi directiva: `using System.Data.OleDb;`

5. Intenţionăm să populăm controlul la încărcarea formei. Prin urmare, tratăm evenimentul **Load** pentru formă. Acţionaţi dublu click pe suprafaţa formei. În corpul *handler*-ului, introduceţi codul marcat cu **Bold**:

```
private void Form1_Load(object sender, EventArgs e)
{
   // Setăm numărul coloanelor şi textul header-elor
   dataGridView1.ColumnCount = 4;
   dataGridView1.Columns[0].HeaderText = "ID-ul";
   dataGridView1.Columns[1].HeaderText = "Numele";
   dataGridView1.Columns[2].HeaderText = "Prenumele";
   dataGridView1.Columns[3].HeaderText = "Data Nasterii";

   // Stringul de conectare
   string connString =
      @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
        C:\teste\Persoane.mdb;";

   // Stringul de interogare
   string sql = @"select * from Persoane";

   // Declarăm variabilele conexiune şi data reader
   OleDbConnection conn = null;
   OleDbDataReader reader = null;

   // Creăm conexiunea
   using (conn = new OleDbConnection(connString))
   {
```

```
            conn.Open();          // Deschidem conexiunea

            // Se execută interogarea
            OleDbCommand cmd = new OleDbCommand(sql, conn);
            reader = cmd.ExecuteReader();

            // Tablou de stringuri. Reține valorile unui rând
            // din tabelă
            string[] r = null;
            // Metoda Read() avansează reader la un rând nou
            while ( reader.Read() )
            {
                r = new string[4]{ reader[0].ToString(),
                                   reader[1].ToString(),
                                   reader[2].ToString(),
                                   reader[3].ToString() };
                // Adăugăm rândul în control
                dataGridView1.Rows.Add(r);
            }
            reader.Close();
        } // using
    }
```

6. Compilați și executați cu **F5**.

Veți avea ceva asemănător:

Form1

ID-ul	Numele	Prenumele	Data Nasterii
1	Voicu	Ionel	16.01.1988 00:0...
2	Grapini	Grigore	10.11.1987 00:0...
3	Pop	Ioan	21.02.1990 00:0...
4	Ulinici	Sorin	14.07.1991 00:0...

De reținut:

- Coloanele, împreună cu textul *header*-elor se pot adăuga *design time*, însă am dorit să arătăm cum le setați programatic.
- Stringul de conectare are două atribute: `Provider` și `Data Source`. Primul indică providerul **OLE DB** pentru bazele de date **Access**, iar al doilea indică sursa de date și locația sa.

- Rândurile se adaugă în control cu metoda `Add()`: `dataGridView1.Rows.Add(r);` , unde `r` este un şir de stringuri care reţine valorile *reader*-ului.
- Declaraţia `using` asigură eliberarea resurselor alocate atât pentru o ieşire normală din bloc, cât şi la aruncarea unei excepţii. Observaţi că nu este nevoie să mai închideţi conexiunea, ci doar ***reader***-ul.
- Remarcăm similitudinea codului şi a tipurilor utilizate pentru *provider*-ii ***SQLServer*** şi **OLE DB**.

Utilizarea provider-ului pentru ODBC

ADO.NET include furnizorul de date **ODBC** în spaţiul de nume **System.Data.Odbc**. Este prima tehnologie Microsoft de accesare a surselor de date, fiind încă folosită pentru surse de date care nu au provider **OLE DB**. Poate fi folosită cu aproape oricare tip de sursă de date. **ODBC** comunică cu sursa de date prin intermediul *driver*-ului sursei de date, deci apare un nivel intermediar şi aceasta face ca viteza de lucru să fie mai mică în raport cu alţi provideri.

Cele mai importante clase asociate provider-ului **ODBC** sunt uşor de recunoscut, după prefixul **Odbc**: `OdbcCommand`, `OdbcConnection`, `OdbcDataAdaptor`, `OdbcDataReader`. Semnificaţia acestora este aceeaşi cu a claselor similare ale celorlalţi provideri.

Vom realiza două aplicaţii. Prima aplicaţie se conectează la un fişier **Excel** iar a doua la un server **MySql**.

Folosim în mod intenţionat metode diferite de conectare **ODBC** pentru cele două aplicaţii: în caul primei aplicaţii includem în stringul de conectare informaţii despre *driver*-ul **Excel**, iar pentru a doua aplicaţie realizăm o sursă de date cu un utilitar integrat sistemului de operare. Precizăm că ambele metode pot fi folosite şi lucrează la fel de bine pentru aplicaţiile cu provider **ODBC**.

Aplicaţia *OdbcExcelExample*

Vom reliza o aplicaţie de tip *Windows Forms* care se conectează prin providerul **ODBC** la un fişier ***Excel 2003***, în care pe prima foaie este un tabel. Datele se citesc din sursa de date şi se afişează într-un control **DataGridView**. Urmaţi indicaţiile:

1. Deschideţi *Micosoft Excel* şi scrieţi tabelul:

	A	B	C	D	E	F
1	IdProdus	Denumire	Pret	Cantitate		
2	1	Paine	2	200		
3	2	Lapte	3	23		
4	3	Iaurt	4	45		
5	4	Cola	5	21		
6						

Salvați fişierul cu numele ***Produse.xls*** în folder-ul **C:\teste**.

2. Creați în **VCSE** un nou proiect de tip *Windows Forms*, cu numele *OdbcExcelExample*.

3. Pe suprafața formei plasați un control de tip **DataGridView**.

4. În fişierul *Form1.cs* introduceți directiva: `using System.Data.Odbc;`

5. Dorim să populăm controlul la încărcarea formei. Deci tratăm evenimentul **Load** pentru formă. Acționați dublu click pe suprafața formei. În corpul *handler*-ului, introduceți codul marcat cu **Bold**:

```
private void Form1_Load(object sender, EventArgs e)
{
    // Setăm numărul coloanelor şi textul header-elor
    dataGridView1.ColumnCount = 4;
    dataGridView1.Columns[0].HeaderText = "ID-ul";
    dataGridView1.Columns[1].HeaderText = "Nume";
    dataGridView1.Columns[2].HeaderText = "Pret";
    dataGridView1.Columns[3].HeaderText = "Nr Produse";

    // Stringul de conectare
    string connString =
            @"Driver={Microsoft Excel Driver (*.xls)};
              DriverId=790;
              Dbq=C:\teste\Produse.xls;
DefaultDir=C:\OdbcExcelExample\OdbcExcelExample\bin\Debug";

    // Stringul de interogare
    // [Foaie1$] poate fi [sheet1$] după caz
    string sql = @"select * from [Foaie1$]";

    // Declarăm variabilele conexiune şi data reader
    OdbcConnection conn = null;
    OdbcDataReader reader = null;

    // Creăm conexiunea
    using (conn = new OdbcConnection(connString))
    {
        conn.Open();          // Deschidem conexiunea
        // Se execută interogarea
        OdbcCommand cmd = new OdbcCommand(sql, conn);
        reader = cmd.ExecuteReader();

        // Tablou de stringuri. Reține valorile unui rând
        // din tabelă
        string[] r = null;
```

```
            // Metoda Read() avansează reader la un rând nou
             while (reader.Read())
             {
                 r = new string[4]{ reader[0].ToString(),
                                    reader[1].ToString(),
                                    reader[2].ToString(),
                                    reader[3].ToString() };
                 dataGridView1.Rows.Add(r);
             }
             reader.Close();
         } // using
    }
```

6. Compilaţi şi rulaţi cu **F5**.

La rulare obţineţi:

Form1

ID-ul	Nume	Pret	Nr Produse
1	Paine	2	200
2	Lapte	3	23
3	Iaurt	4	45
4	Cola	5	21

Observaţii:

- Stringul de conectare precizează driver-ul *Excel* şi ID-ul său. **Dbq** indică sursa de date şi calea de director, iar **DefaultDir** este directorul implicit.
- Codul C# este similar cu cel pentru provider-ii **OLE DB** sau **SQL Server**.

Aplicaţia *OdbcMySQL*

Scopul acestei aplicaţii este acela de a arăta modul în care vă puteţi conecta la o bază de date **MySQL** cu C#, utilizând provider-ul **ODBC**.

Diferenţa estenţială faţă de modul de conectare al aplicaţiei anterioare constă în faptul că vom construi un ***Data Source Name*** (***DSN***) cu ajutorul unui utilitar windows. Acest *DSN* se utilizează apoi în stringul de conectare.

Separăm tema în trei secţiuni:

A) Instalarea serverului MySQL 5.0 şi crearea unei baze de date MySQL

Presupunând că nu aveţi **MySQL** instalat şi că nu aţi mai lucrat cu această bază de date, procedaţi după cum urmează:

- Intrați pe site-ul **www.mysql.com**. La secțiunea *download*, descărcați fişierul **mysql-essential-5.0.45-win32.msi**.
- *Wizard*-ul de instalare vă cere anumite informații, pe care le completați selectând următoarele opțiuni (bifați *check box*-urile):
  - ✓ "*Configure the MySQL Server now*".
  - ✓ "Install as Windows Service".
  - ✓ "*Launch the MySQL Server Automatically*".
  - ✓ "*Include Bin Directory in Windows PATH*".
  - ✓ "*Modify Security Settings*". Introduceți parola de **root** în cele două *text box*-uri.
  - ✓ "*Enable root access from remote machines*".
  - ✓ "*Create An Anonymous Account*".

În felul acesta, ați instalat **MySQL Server 5.0** ca **serviciu Windows** care startează în mod automat la pornirea calculatorului. Aveți un cont de administrator (**root**) şi un cont ***Anonymous***. Acesta din urmă vă permite să vă conectați fără să introduceți user sau parolă. Este bine să aveți un asemenea *user* numai în perioada de învățare, deoarece în practică creează probleme de securitate.

Acum serverul e lansat. Îl putem folosi. Ne conectăm cu clientul **mysql.exe** la serverul **MySQL** pentru a crea o bază de date cu o tabelă. Urmați indicațiile :

1. În *taskbar*-ul sistemului de operare, click *Start->Run*. Introduceți **cmd** şi apăsați **OK**.
2. Tastați la promptul sistemului: `mysql`. În acest fel v-ați conectat ca user *Anonymous*, deoarece nu ați specificat user şi parolă.
3. Tastați **`create database clasa;`**
   Ați construit astfel o bază de date cu numele **`clasa`**.
4. Tastați **`show databases;`**
   Vizualizați astfel bazele de date preinstalete şi noua bază de date creată.
5. Introduceți comanda **`use clasa;`** pentru a utiliza această bază de date.

```
C:\WINDOWS\system32\cmd.exe - mysql

C:\>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.0.45-community-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database clasa;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| clasa              |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)

mysql> use clasa;
Database changed
mysql>
```

6. Creăm o tabelă. Introduceți comanda:

```
create table Elevi
(Nume varchar(15), Prenume varchar(15), Varsta int);
```

7. Inserăm câteva rânduri în tabelă:

```
insert into Elevi values ('Blaga', 'Ionel', 25);
insert into Elevi values ('Horga', 'Lucian', 25);
insert into Elevi values ('Chifa', 'Ionel', 19);
```

```
C:\WINDOWS\system32\cmd.exe - mysql

mysql> create table Elevi (Nume varchar(15), Prenume varchar(15), Varsta int);
Query OK, 0 rows affected (0.08 sec)

mysql> insert into Elevi values ('Blaga', 'Ionel', 25);
Query OK, 1 row affected (0.05 sec)

mysql> insert into Elevi values ('Horga', 'Lucian', 25);
Query OK, 1 row affected (0.01 sec)

mysql> insert into Elevi values ('Chifa', 'Ionel', 19);
Query OK, 1 row affected (0.01 sec)

mysql> select * from Elevi;
+-------+---------+--------+
| Nume  | Prenume | Varsta |
+-------+---------+--------+
| Blaga | Ionel   |     25 |
| Horga | Lucian  |     25 |
| Chifa | Ionel   |     19 |
+-------+---------+--------+
3 rows in set (0.02 sec)

mysql>
```

8. Acum avem baza de date **`Clase`** cu o tabelă numită **`Elevi`**. Tabela conține trei rânduri. Vă deconectați de la server cu comanda `quit`.

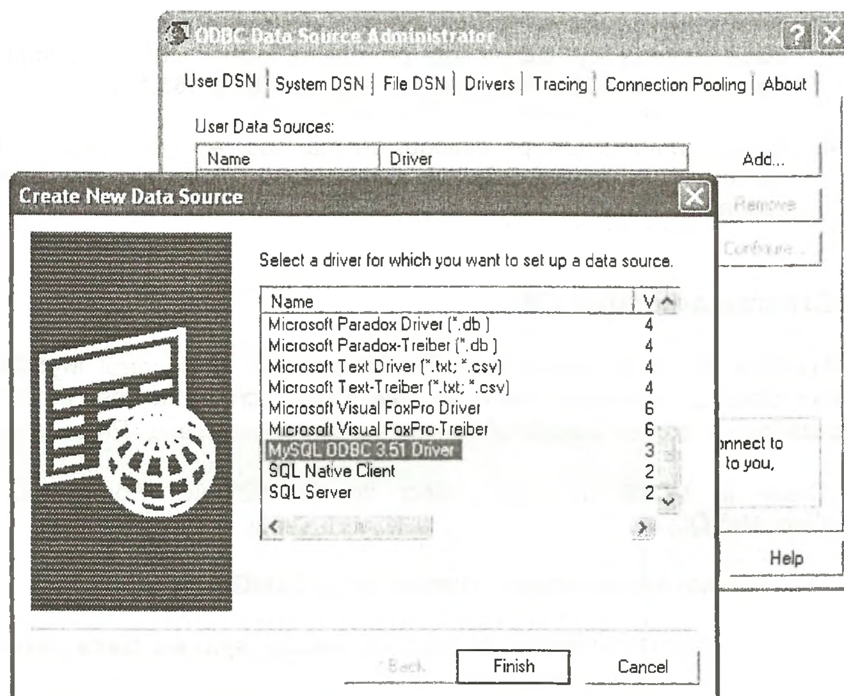
B) Crearea unui *Data Source Name* (DSN)

Informațiile necesare conectării la o sursă de date cu provider **ODBC** se pot centraliza cu utilitarul *ODBC Data Source Administrator*. Ceea ce se obține se numeşte *Data Source Name* (DSN).

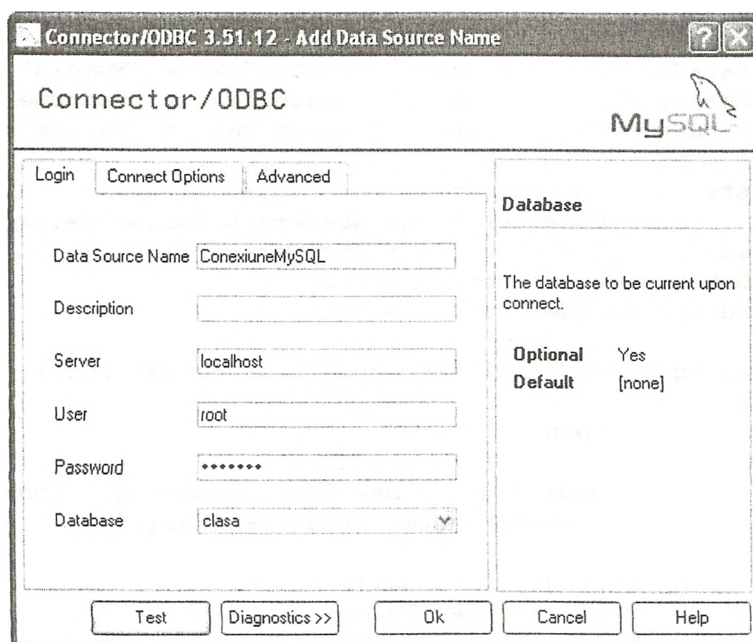
Procedați astfel:

1. În bara de task-uri a sistemului de operare, accesați *Start->All Programs->Control Panel*. Dublu click pe iconul *Administrative Tools*, apoi lansați *Data Sources(ODBC)*. Se deschide dialogul *ODBC Data Source Administrator*.

2. În pagina *User DSN*, apăsați butonul *Add*. Se deschide dialogul *Create New Data Source*. Selectați din listă *driver*-ul **MySQL**, apoi click pe butonul *Finish*:



3. Completați formularul de mai jos. În câmpul *Data Source Name*, introduceți *ConexiuneMySQL*. Dacă doriți să vă conectați ca user **Anonymous**, nu este necesar să completați câmpurile **User** şi **Password**.



Dacă serverul **MySQL** nu este pe maşina locală, atunci în câmpul *Server* treceţi **IP**-ul *host*-ului respectiv. De exemplu: **80.97.66.192**

4. Apăsaţi butonul **OK** pe dialogul de mai sus. Aţi creat deja un **DSN** cu numele *ConexiuneMySQL*, pe care îl pasăm stringului de conectare în aplicaţia C# care urmează.

C) Crearea aplicaţiei C#

Aplicaţia de tip *Windows Forms* se conectează la serverul **MySQL** de pe maşina locală şi accesează baza de date **clasa**, creată la paşii anteriori. Apoi afişează într-un control **DataGridView** datele din tabela **Elevi**. Procedaţi astfel:

1. Creaţi în **VCSE** un nou proiect de tip *Windows Forms*, cu numele *OdbcMySQL*.

2. Pe suprafaţa formei plasaţi un control de tip **DataGridView**.

3. În fişierul *Form1.cs* introduceţi directiva: **using System.Data.Odbc;**

4. Controlul se va popula la încărcarea formei. Prin urmare, tratăm evenimentul **Load** pentru formă. Acţionaţi dublu click pe suprafaţa formei. În corpul *handler*-ului, introduceţi codul marcat cu **Bold**:

```
private void Form1_Load(object sender, EventArgs e)
{
    dataGridView1.ColumnCount = 3;
    dataGridView1.Columns[0].HeaderText = "Numele";
    dataGridView1.Columns[1].HeaderText = "Prenumele";
    dataGridView1.Columns[2].HeaderText = "Varsta";

    string connString =
        "dsn=ConexiuneMySQL;User=root;Password=iuliuta";
    string sql = @"select * from elevi";
    OdbcConnection conn = null;
    OdbcDataReader reader = null;

    using (conn = new OdbcConnection(connString))
    {
        conn.Open();
        // Se execută interogarea
        OdbcCommand cmd = new OdbcCommand(sql, conn);
        using (reader = cmd.ExecuteReader())
        {
           string[] r = null;
           while (reader.Read())
           {
```

```
                    r = new string[3]{ reader[0].ToString(),
                                       reader[1].ToString(),
                                       reader[2].ToString() };
                    dataGridView1.Rows.Add(r);
                }
            }
        }
    }
```

5. Compilați şi executați cu **F5**.

La rulare, controlul **DataGridView** afişează datele din tabela **`Elevi`**, obținute în urma interogării **SELECT**:

Form1

Numele	Prenumele	Varsta
Blaga	Ionel	25
Horga	Lucian	25
Chifa	Ionel	19

Precizări:

- **MySQL** oferă propriul provider **ODBC**, numit **MyODBC**. Este driver-ul pe care l-am utilizat la crearea **DSN**.
- **ADO.NET** are un provider dedicat pentru **MySQL**, definit în spațiul de nume **MySql.Data.MySqlClient**.
- Nu am comentat codul, deoarece este similar cu cel al aplicațiilor anterioare cu provider ODBC.

Capitolul 12

Aplicații cu baze de date în modelul deconectat

Dacă ceea ce doriți este doar să citiți și să afișați mari cantități de date, atunci un obiect de tip **DataReader** este foarte potrivit. În cazul în care trebuie să efectuați mai multe operații asupra acestor date și la final să actualizați baza de date, atunci un *reader* nu este cea mai potrivită alegere, deoarece el parcurge datele o singură dată, doar înainte și pentru fiecare operație este nevoie să construiți un alt obiect de tip **DataReader**. Ceea ce trebuie să faceți în aceste situații, este să folosiți un *dataset*.

Un *dataset* este un obiect de tip **System.Data.DataSet**. *Dataset*-urile sunt capabile să depoziteze mari cantități de date în memoria internă a calculatorului. Un *dataset* vă permite să preluați date din sursa de date, să închideți conexiunea cu baza de date, apoi să prelucrați datele *offline*, adică în modul deconectat de la sursa de date. La final, când prelucrarea datelor s-a încheiat, baza de date se actualizează.

Un *dataset* memorează datele într-o colecție de tabele (obiecte de tip **DataTable**) și de relații între tabele.

În momentul în care construiți un obiect de tip **DataSet**, acesta nu conține date. Ca să primească date este nevoie de un obiect de tip **DataAdapter**. Un **DataAdapter** este parte a furnizorului de date, în vreme ce un *dataset* este exterior provider-ului. Fiecare provider are propriul său adaptor de date. Spre exemplu, adaptorul provider-ului pentru ***SQL Server*** este **SqlDataAdaptor**.

Construirea și utilizarea dataset-urilor

Ca să creați o instanță a clasei **DataSet**, utilizați unul dintre constructorii clasei:

```
DataSet ds = new DataSet();
DataSet ds = new DataSet("numeDataSet");
```

Al doilea constructor precizează numele dataset-ului. Primul constructor, atribuie *dataset*-ului numele *NewDataSet*.

În *dataset*, datele sunt organizate într-un set de tabele. Fiecare tabelă se memorează într-un obiect de tip **DataTable**. Fiecare obiect **DataTable** consistă din obiecte de tip **DataRow** și **DataColumn**, care păstrează datele.

Să presupunem că un *dataset* primește datele de la un **SqlDataAdaptor**. Secvența standard de cod care construiește și umple *dataset*-ul este :

```
// Creează adaptorul de date
SqlDataAdapter da = new SqlDataAdapter(sql, conn);
```

```
// Creează dataset-ul
DataSet ds = new DataSet();

// Încarcă dataset-ul cu tabela elevi. Dacă există şi alte
// tabele în sursa de date, acelea nu se încarcă
da.Fill(ds, "elevi");
```

Aşadar, constructorul clasei **SqlDataAdaptor** primeşte stringul de conectare şi conexiunea (referința la un obiect de tip **SqlConnection**), iar metoda **Fill** a adaptorului umple *dataset*-ul cu tabela "**elevi**".
Există încă două versiuni ale constructoului clasei SqlDataAdaptor. Primul nu are parametri, iar al doilea primeşte un obiect de tip **SqlCommand**:

```
SqlDataAdapter da = new SqlDataAdapter();
SqlDataAdapter da = new SqlDataAdapter(cmd);
```

Pentru a încărca în *dataset* toate tabelele din baza de date, utilizați altă versiune a metodei **Fill**:

```
da.Fill(ds);
```

Accesarea tabelelor într-un dataset

Un *dataset* poate să conțină mai multe tabele. Acestea sunt obiecte de tip **DataTable** care se pot accesa indexat, începând cu indexul **0**. de exemplu :

```
// Obține tabela a treia din dataset.
DataTable dt = ds.Tables[2] ;
```

Altă variantă este accesarea prin numele tabelei:

```
// Obține tabela produse
DataTable dt = ds.Tables["produse"];
```

Accesarea rândurilor şi coloanelor într-o tabelă

Rândurile unui unui obiect **DataTable** se obțin din proprietatea **Rows**, care suportă indexarea :

```
// Obține al patrulea rând al tablelei
DataRow r = dt.Rows[3];
```

Colecția de coloane a unui obiect **DataTable** se poate obține din proprietatea **Columns**. Proprietatea suportă operatorul de indexare, cu pornire de la 0:

```
// Obţine coloana a 4-a
DataColumn col = dt.Columns[3];

// Obţine coloana "Nume"
DataColumn col = dt.Columns["Nume"];
```

Accesarea valorilor dintr-o tabelă a unui dataset

Se utilizează proprietăţile `Rows` şi `Columns` ale clasei `DataTable`:

```
// Obţinem tabela dorită
DataTable dt = ds.Tables["produse"] ;

// Pentru fiecare rând din colecţie (din tabela produse)
foreach (DataRow row in dt.Rows)
{
    // Pentru fiecare coloană
    foreach (DataColumn col in dt.Columns)
        Console.Write(row[col]);  // Accesare indexată a
                                  // valorii unei celule
     Console.WriteLine();
}
```

Propagarea schimbărilor din dataset spre baza de date

Am văzut că metoda `Fill()` a clasei `TableAdaptor` umple un *dataset* cu datele preluate din baza de date. Dacă metoda este apelată după ce s-a închis conexiunea, atunci `Fill()` deschide conexiunea, transferă datele în dataset şi apoi închide conexiunea. Dacă conexiunea era deschisă înainte de apelul `Fill()`, atunci metoda lasă conexiunea deschisă.

Modificările operate asupra datelor unui *dataset*, se propagă înapoi în baza de date la apelul metodei `Update()` a clasei `TableAdaptor`. Prezentăm două versiuni ale acestei metode. Ambele returnează numărul de rânduri updatate cu succes.

Prima versiune actualizează baza de date operând toate modificările care s-au făcut în *dataset*:

```
int Update(DataSet ds);
```

A doua versiune actualizează baza de date operând toate modificările care s-au făcut în tabela transmisă ca argument:

```
int Update(DataTable dt)
```

Actualizarea bazei de date se produce conform scenariului de mai jos:

```
// Creează adaptorul de date
SqlDataAdapter da = new SqlDataAdapter(sql, conn);

// Creează dataset-ul
DataSet ds = new DataSet();

// Umple dataset-ul cu date din baza de date
da.Fill(ds);

// Se modifică datele în dataset
// ...

// Se propagă modificările în baza de date
ta.Update(ds);
```

Aplicația *UpdateDatabase*

Aplicația de tip *Windows Forms* accesează o bază de date ***SQL Server 2005*** şi afişează într-un control `DataGridView` datele dintr-o tabelă. La apăsarea unui buton, modificările operate de utilizator în celulele gridului se propagă în baza de date.

1. Creați în **VCSE** un nou proiect de tip *Windows Forms*, cu numele *UpdateDatabase*.

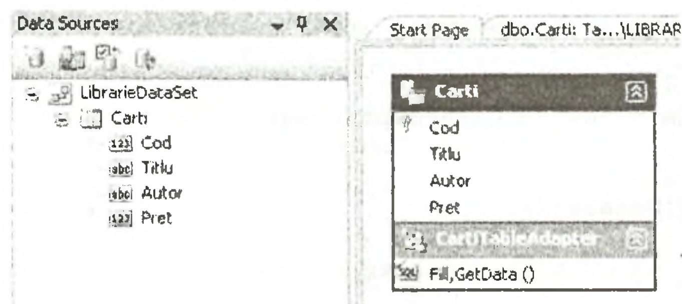
2. Pe suprafața formei plasați un control de tip **DataGridView** şi un buton cu textul *Salveaza* şi o etichetă cu textul "*0 randuri afectate*".

3. Adăugați proiectului o bază de date cu numele ***Librarie***. În acest scop, urmați indicațiile subcapitolului "**Crearea unei baze de date in VCSE**". *Wizardul* propune numele `LibrarieDataset` pentru clasa pe care o generează. Această clasă moşteneşte `DataSet`, aşadar **Este Un** *dataset*.

4. Adăugați bazei de date ***Librarie*** tabela `Carti`, cu următoarea structură:

Column Name	Data Type	Allow Nulls
Cod	int	☐
Titlu	nvarchar(50)	☐
Autor	nvarchar(20)	☑
Pret	int	☑

5. În fereastra ***Data Source***, acționați click dreapta pe tabela `Carti` şi selectați *Edit DataSet with Designer*. După ce *designer*-ul se deschide, trageți tabela `Carti` cu mouse-ul, din fereastra ***Data Source*** pe suprafața designerului:



Prin această acțiune am cerut ajutorul mediului integrat, care a generat pentru noi o clasă **CartiTableAdapter**. Această clasă conține un obiect de tip **SqlDataAdapter**, un obiect de tip **SqlConnection** și un obiect de tip **SqlCommand**, astfel încât nu mai trebuie să ne procupăm de deschiderea manuală a unei conexiuni la server.

6. Deschideți în *Editorul de Cod* fișierul *Form1.cs*. În clasa **Form1** declarați câmpurile:

```
// Declarăm o referință la dataset
private LibrarieDataSet libDataSet = null;

// Referință la un obiect de tip CartiTableAdapter
private LibrarieDataSetTableAdapters.CartiTableAdapter
                                          ta = null;
```

7. Populăm controlul **DataGridView** la încărcarea formei. Deci tratăm evenimentul **Load** pentru formă. Acționați dublu click pe suprafața formei. În corpul *handler*-ului, introduceți codul marcat cu **Bold**:

```
private void Form1_Load(object sender, EventArgs e)
{
    // Instanțiem adaptorul de date. Observați că este
    // necesar să specificăm spațiul de nume în care este
    // definită clasa CartiTableAdaptor
    ta = new
        LibrarieDataSetTableAdapters.CartiTableAdapter();

    // Creăm un dataset
    libDataSet = new LibrarieDataSet();

    // Umplem dataset-ul cu datele din tabela Carti
    ta.Fill(libDataSet.Carti);

    // bs permite legarea controlului dataGridView1
    // la sursa de date din dataset
    BindingSource bs = new BindingSource(libDataSet,
                                        "Carti");
```

```
        // Legarea se face prin proprietatea DataSource
        dataGridView1.DataSource = bs;
    }
```

8. La acționarea butonului *Salveaza*, se transmit modificările din *dataset* în baza de date. Tratăm evenimentul **Click**. Acționați dublu click pe buton. Metoda de tratare se editează astfel:

```
private void button1_Click(object sender, EventArgs e)
{
    // Propagăm modificările spre baza de date.
    int afectate = ta.Update(libDataSet.Carti);
    label1.Text = afectate + " randuri afectate";
}
```

9. Compilați şi rulați cu **F5**.

Aplicația afişează forma:

Form1

0 randuri afectate   Salveaza

Cod	Titlu	Autor	Pret
231	Ion	Liviu Rebreanu	32
1437	Rosu si Negru	Stendhal	33
3512	Quo Vadis	Sienkiewicz	35
6423	Ana KArenina	Tolstoi	48

Dacă spre exemplu operăm modificări în celulele din randurile **3** şi **4**, iar apoi apăsăm *Salveaza*, obținem:

Form1

2 randuri afectate   Salveaza

Cod	Titlu	Autor	Pret
231	Ion	Liviu Rebreanu	32
1437	Rosu si Negru	Stendhal	33
3512	Quo Vadis	Henrik Sienkiewicz	35
6423	Ana Karenina	Lev Tolstoi	48

La o nouă rulare cu **F5** veți constata că modificările aduse bazei de date sunt persistente.

Observații:

- Mediul VCSE creează cele două fişiere ale bazei de date (**.mdf** şi **.LDF**) în folderul de bază al aplicației, la un loc cu fişierul **.csproj**. În momentul în care faceți ***Build*** cu **F5** (modul depanare), se face o copie a celor două fişiere în folderul **\Bin\Debug**, iar aplicația se lansează şi lucrează cu aceste cópii. La noi rulări cu **F5**, nu se mai face ***Build*** dacă nu ați adus modificări în cod, ci numai lansări în execuție. Constatați că baza de date din **\Bin\Debug** se actualizează. Dacă ați operat modificări în cod, atunci la un **F5** se face şi ***Build*** şi baza de date originală se copiază peste cea din **\Bin\Debug**.
- Stringul de conectare se generează în mod automat la crearea bazei de date cu ajutorul mediului integrat. Se poate vizualiza astfel: în ***Solution Explorer***, click dreapta pe fişierul *app.config* şi selectați *Open*. Veți constata informațiile de conectare se păstrează în format **XML**.
- Atribuirea `dataGridView1.DataSource = bs;` ***leagă*** controlul de *dataset*, astfel încât orice modificare operată in celulele controlului, se transmite în mod automat tabelei din *dataset*.

Dataset-urile şi XML

Când lucrați cu *dataset*-uri, aveți posibilitatea să salvați datele local, într-un fişier de tip **XML**, astfel încât informațiile să persiste şi în alt loc decât în baza de date după încheierea sesiunii.
Salvați datele cu ajutorul metodei `WriteXml()` a clasei `DataSet` astfel:

```
ds.WriteXml(@"C:\date.xml");
```

Când deschideți o nouă sesiune a aplicației, aveți alternativa de a reîncărca în *dataset* datele preluate din fişierul XML. Citirea se face cu metoda `ReadXml()`:

```
ds.ReadXML("C:\date.xml");
```

Exemplu:

Ne întoarcem la aplicația *UpdateDatabase*, realizată anterior. Introduceți ca ultimă linie în corpul metodei de tratare a evenimentului **Click** pentru butonul *Salveaza*, apelul:

```
libDataSet.WriteXml(@"C:\carti_bune.xml");
```

Rulați cu **F5**, apoi căutați fişierul `carti_bune.xml`. Veți constata că toate informațiile din *dataset* s-au salvat în format **XML**.

Controalele şi legarea datelor

Anumite proprietăţi ale controalelor se pot lega la o sursă de date. Sursele de date sunt diverse: o proprietate a altui control, o celulă, un rând sau o tabelă într-un dataset, sau o simplă variabilă. Conceptul se numeşte *Data Binding*. După ce au fost legate, valorile din sursa de date schimbă valorile din proprietatea legată şi invers.

Controalele *Windows Forms* suportă două tipuri de legare a datelor (*data binding*):

- **Legarea simplă.**
- **Legarea complexă.**

Legarea simplă a datelor

Acest tip de legare vă permite să ataşaţi o proprietate a unui control la **o singură** valoare din sursa de date. Este utilă pentru controale ca **TextBox** sau **Label**, care afişează o singură valoare. Aplicaţia următoare arată cum puteţi lega proprietăţile a două controale.

Aplicaţia *SimpleDataBinding*

1. Creaţi un nou proiect *WindowsForms* cu numele *SimpleDataBinding*.

2. Pe suprafaţa formei plasaţi două controale: un **`TextBox`** şi un **Label**.

3. În constructorul clasei *Form1*, introduceţi codul evidenţiat în **Bold**:

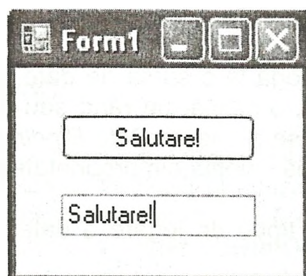
```
public Form1()
{
    InitializeComponent();
    // Legăm proprietatea Text a butonului (primul
    // parametru) de proprietatea Text a text box-ului
    button1.DataBindings.Add("Text", textBox1, "Text");
}
```

4. Tratăm evenimentul **Click** al butonului. Acţionaţi dublu click pe buton şi introduceţi codul marcat în **Bold**, în metoda de tratare:

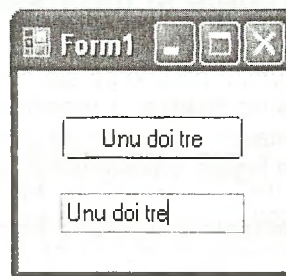
```
private void button1_Click(object sender, EventArgs e)
{
    button1.Text = "Salutare!";
}
```

5. Rulaţi cu **F5**.

Faceţi click pe buton, apoi un click în *text box* ca să primească focusul. În *text box* se afişează "**`Salutare!`**". Dacă introduceţi text în *textbox*, eticheta butonului se modifică corespunzător.



Click pe buton



Editare în text box

Legarea complexă a datelor

Aplicaţi acest tip de legare atunci când vreţi să afişaţi o listă de valori din sursa de date. Controalele care suportă legarea complexă sunt **DataGridView**, **ListBox**, **ComboBox**, **CheckedListBox**. Aceste controale au de proprietatățile DataSource şi DataMember, cu ajutorul cărora le puteţi lega la tabela unei baze de date, astfel:

```
dataGridView1.DataSource = librarieDataSet;
dataGridView1.DataMember = "carti";
```

În exemplu, proprietății `DataSource` a gridului i se atribuie *dataset*-ul aplicației, iar proprietății `DataMember` i se atribuie numele unei tabele din *dataset*. Aceste setări se pot face uşor în fereastra ***Properties***, aşa cum vom vedea în continuare.

Aplicația *ComplexDataBinding1*

Aplicația utilizează utilizează facilitățile mediului integrat pentru a lega proprietatățile unui control **DataGridView** la o tabelă a unei baze de date *SQL Server 2005 Express*. Urmaţi paşii:

1. Creaţi în **VCSE** un nou proiect de tip *Windows Forms*, cu numele *ComplexDataBinding1*.

2. Adăugați proiectului o bază de date cu numele ***Spectacole***. În acest scop, urmaţi indicațiile subcapitolului "**Crearea unei baze de date in VCSE**". *Wizardul* propune numele `SpectacoleDataSet` pentru clasa pe care o generează. Acceptaţi acest nume.

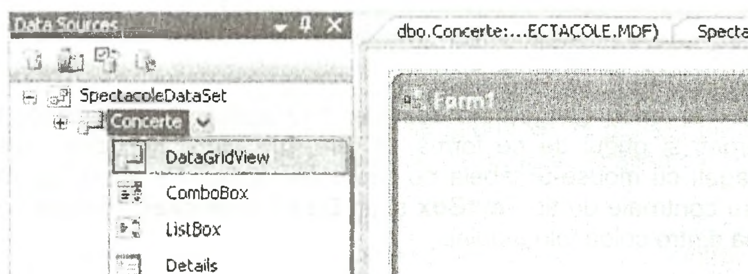
3. Adăugaţi bazei de date ***Spectacole*** tabela `Concerte`, cu următoarea structură:

Column Name	Data Type	Allow Nulls
IDConcert	int	☐
TipConcert	nvarchar(20)	☑
SustinutDe	nvarchar(50)	☐
Data	datetime	☐
Locatie	nvarchar(20)	☑

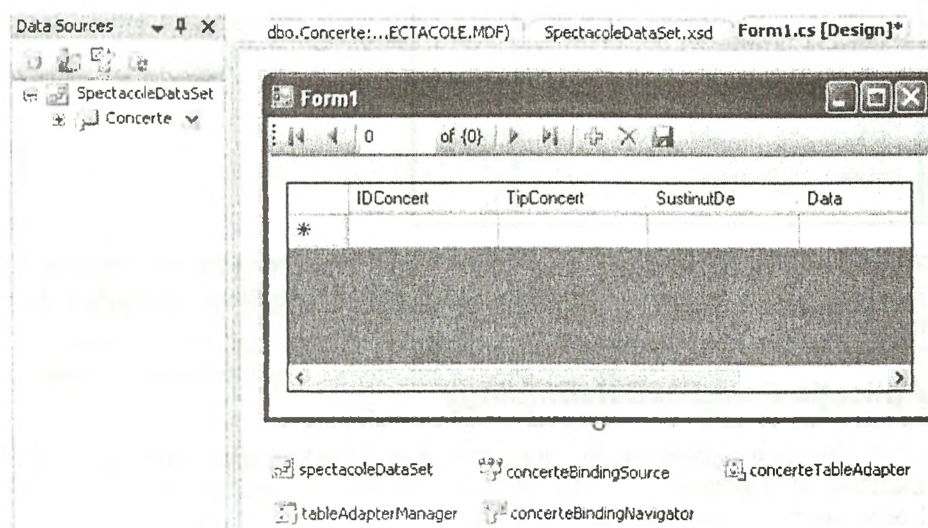
Completați câteva rânduri cu date în tabelă.

4. În fereastra ***Data Source*** acționați click dreapta pe tabela **`Concerte`** şi selectați *Edit DataSet with Designer*. După ce *designer*-ul se deschide, trageți tabela **`Concerte`** cu mouse-ul din fereastra ***Data Source*** pe suprafața designerului. Prin această acțiune am cerut ajutorul mediului integrat, care a generat pentru noi o clasă **`ConcerteTableAdapter`**.

5. În *View Designer* selectați forma, iar în fereastra ***Data Sources*** deschideți lista ataşată tabelei **`Concerte`** şi selectați *DataGridView*.



6. Urmează o acțiune simplă: în fereastra ***Data Sources*** selectați cu mouse-ul tabela **`Concerte`**, apoi o trageți peste formă. Pe suprafața formei apare un control **DataGridView**, cu celulele legate la tabela **`Concerte`**:



Observați apariția în *designer tray* a câtorva referințe la obiecte de tip **`TableAdaptor`** sau **`BindingNavigator`**.

7. Rulați cu **F5**.

Veți obține:

Form1

1 of 4

IDConcert	TipConcert	SustinutDe	Data
1	Rock	Metallica	12.03.2009
2	Hip Hop	Eminem	04.07.2009
3	House	Mickey Oliver	15.11.2009
4	Country	Martina McBride	24.02.2009

Încercați singuri:

În proiectul anterior, ştergeți din *designer tray* apăsând tasta *delete* toate referințele. Ştergeți şi gridul de pe formă. În fereastra ***Data Sources***, selectați *Details*, apoi trageți cu mouse-ul tabela pe suprafața formei. Pe suprafața formei vor apărea patru controale de tip **TextBox** şi un **DateTimePicker**. Fiecare control este legat la una dintre coloanele tabelei.

Form1

2 of 4

IDConcert: 2

Tip Concert: Hip Hop

Sustinut De: Eminem

Data: 4 iulie 2009

Locatie: Bucuresti

Toate controalele indică acelaşi rând din tabelă. Navigarea de la un rând la altul se face cu ajutorul butoanelor săgeată din controlul **ToolStrip** din partea de sus a formei.

Aplicația *ComplexDataBinding2*

Modificăm proiectul anterior astfel încât să legăm proprietăți ale controalelor **ListBox** şi **TextBox** la baza de date, ca să fie posibilă editarea datelor şi trasmiterea modificărilor bazei de date.

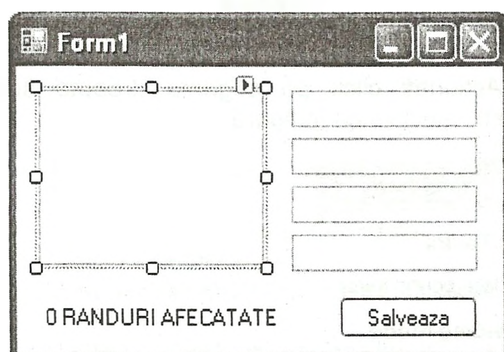
Paşii **1**, **2**, **3**, **4** sunt identici cu cei ai aplicației precedente. Schimbați doar numele proiectului : *ComplexDataBinding2*.

5. În *Dataset Designer*, acționați click dreapta pe itemul **ConcerteTableAdapter** şi selectați *Configure*...

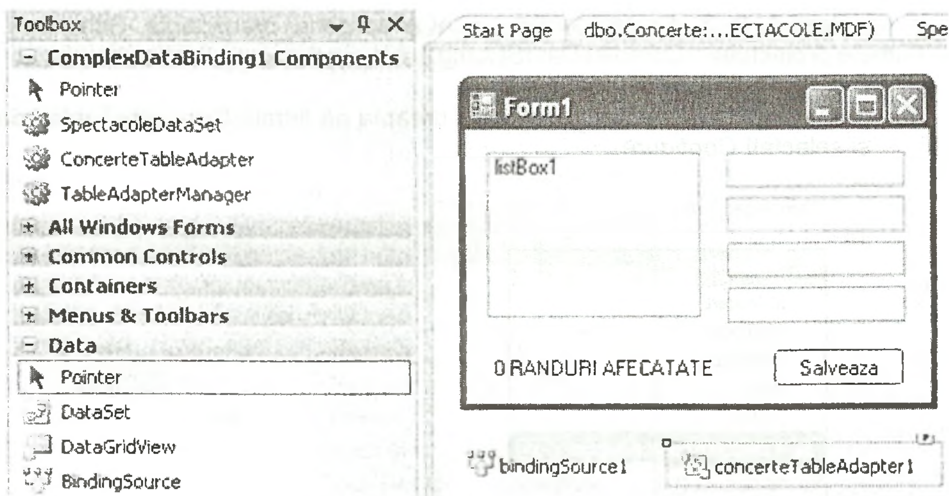


Configurăm adaptorul, deoarece vom apela metoda `Update()` pentru actualizarea bazei de date şi această metodă utilizează interogări **SQL** în acest scop. Vom genera interogările în mod automat. Apăsați butonul *Next* al primului dialog. Al doilea dialog ne spune că va genera pentru noi metodele `Fill`, `GetData`, `Insert`, `Delete` şi `Update`. Apăsați din nou *Next*, apoi pe ultimul dialog, *Finish*.

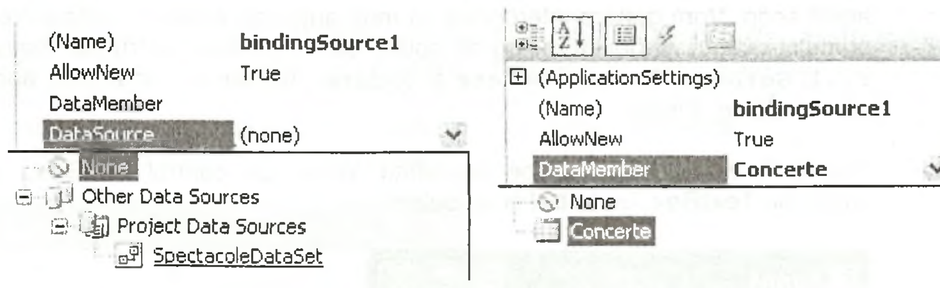
6. Plasați în *Form Designer* pe suprafața formei un control **ListBox**, patru controale **TextBox**, un **Label** şi un buton :



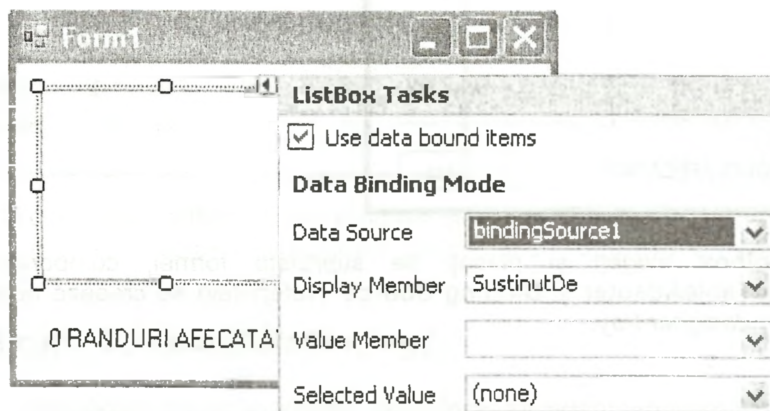
7. Din **Toolbox** alegeți şi plasați pe suprafața formei, componentele: **ConcerteTableAdapter** şi **Binding Source**. Referințele se creează automat şi apar în *designer tray*.



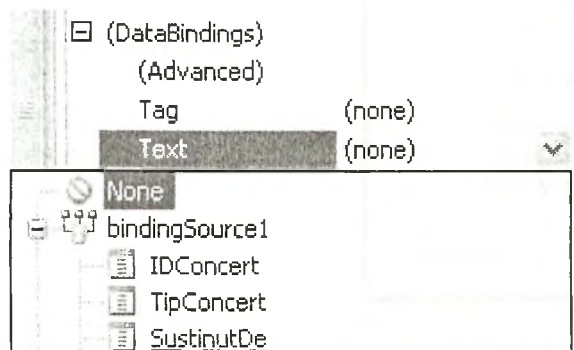
8. Selectați în *designer tray* `bindingSource1`. În fereastra ***Properties*** atribuiți proprietății `DataSource` valoarea `SpectacoleDataSet`, iar proprietății `DataMember`, tabela `Concerte`:



9. Selectați controlul **ListBox**. Acționați click pe săgeata dreapta sus. În fereastra care se deschide, faceți selecțiile ca în figură:



10. Selectați controlul **TextBox** cel mai de sus. În fereastra ***Properties***, expandați itemul **Data Bindings** și setați proprietatea `Text` la valoarea `SustinutDe` :



11. Executați aceleași operații de la pasul **10**, pentru următoarele trei ***textbox***-uri, cu diferența că veți lega proprietatea `Text` la celelalte coloane ale tabelei.

12. La încărcarea formei, vom umple *dataset*-ul. Dublu click pe suprafața formei. În *Editorul de Cod* scrieți:

```
private void Form1_Load(object sender, EventArgs e)
{
  // Fill() încarcă datele din baza de date în dataset
  concerteTableAdapter1.Fill(spectacoleDataSet.Concerte);
}
```

13. La click pe butonul *Salvează*, transmitem bazei de date modificările făcute de utilizator. Dublu click pe buton. Scrieți în metoda *handler*:

```
private void button1_Click(object sender, EventArgs e)
{
   // Aplicăm sursei de date modificările făcute
   bindingSource1.EndEdit();

   // Updatăm baza de date
   int afectate =
   concerteTableAdapter1.Update(spectacoleDataSet);

   label1.Text = afectate + " randuri afectate";
}
```

14. Rulați cu **F5**.

Se obține:

La click în *list box*, textele se modifică în controalele text box în mod automat, astfel încât toate controalele de pe formă indică acelaşi rând din tabelă. Valorile celulelor tabelei se pot edita din *text box*-uri, iar modificările în baza de date sunt persistente.

Capitolul 13

Relaţii între tabele

Între tabelele unei baze de date (şi a unui *dataset*) pot să existe relaţii de tip ***părinte-copil***. Să presupunem că în baza de date **Scoala.mdf** există două tabele: **`Clase`** şi **`Elevi`**. O clasă are mai mulţi elevi. Construim deci o relaţie între cele două tabele, de tip ***una la mai multe*** (*one to many*).

Constrângerea Cheie Străină-Cheie Primară

În tabela **`Clase`** avem un câmp **`Clasa`**, care reprezintă numele clasei (ex. XII-B). Îl setăm cheie primară. În tabela **`Elevi`** introducem un câmp cu acelaşi nume. Nu dorim să existe elevi pentru care câmpul **`Clasa`** are o valoare care nu există în tabela **`Clase`**. Impunem o constrângere de tip ***foreign key-primary key***. Este o *constrângere referenţială*. **`Clase`** este tabela părinte, iar **`Elevi`**, tabela copil. Un rând al tabelei părinte poate să refere multiple rânduri în tabela copil.

Aplicaţia *Scoala*

Aplicaţia creează baza de date şi tabelele precizate mai sus, impunând o constrângere de *cheie străină-cheie primară*. Dacă un rând în tabela **`Clase`** se şterge, atunci dorim să se şteargă toate rândurile corespunzătoare din tabela **`Elevi`** pentru a păstra ***integritatea datelor***. Vom seta regula de ştergere pentru această relaţie: **ÎN CASCADĂ**.

Atenţie !

Vă sugerăm să nu ocoliţi acest proiect, deoarece pe structura lui vom lucra în temele următoare, unde va fi dezvoltat şi completat.

1. Realizaţi în **VCSE** un proiect de tip *Windows Forms*, cu numele *Scoala*.

2. Adăugaţi proiectului o bază de date cu numele ***Scoala***. În acest scop, urmaţi indicaţiile subcapitolului "**Crearea unei baze de date in VCSE**". Acceptaţi numele **`ScoalaDataSet`** pentru clasa de tip **`DataSet`** care se generează.

3. Adăugaţi bazei de date ***Scoala*** tabelele **`Clase`** şi **`Elevi`** cu următoarele structuri:

Tabela **`Clase`**

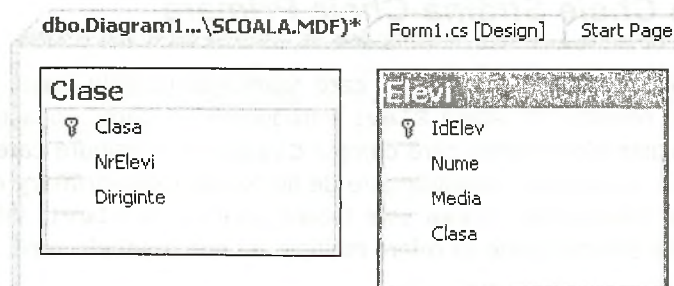
Column Name	Data Type	Allow Nulls
Clasa	nvarchar(50)	☐
NrElevi	int	☐
Diriginte	nvarchar(50)	☑

Tabela **`Elevi`**

	Column Name	Data Type	Allow Nulls
🔑	IdElev	int	☐
	Nume	nvarchar(50)	☑
	Media	real	☑

După completarea schemei, introduceți câteva rânduri în fiecare tabelă.

4. În ***Database Explorer***, click dreapta pe *Database Diagrams*. În dialogul *Add Table* selectați pe rând tabelele **`Clase`** și **`Elevi`** și apăsați butonul *Add*. Diagramele celor două tabele apar în *designer*.



5. Introducem constrângerea de cheie străină. Punctați cu mouse-ul în câmpul **`Clasa`** din tabela **`Clase`** și trageți ținând butonul apăsat până pe câmpul **`Clasa`** al tabelei **`Elevi`**. Se deschide un dialog care identifică cheia primară, cheia străină și numele relației care se crează:

Tables and Columns

Relationship name:

FK_Elevi_Clase

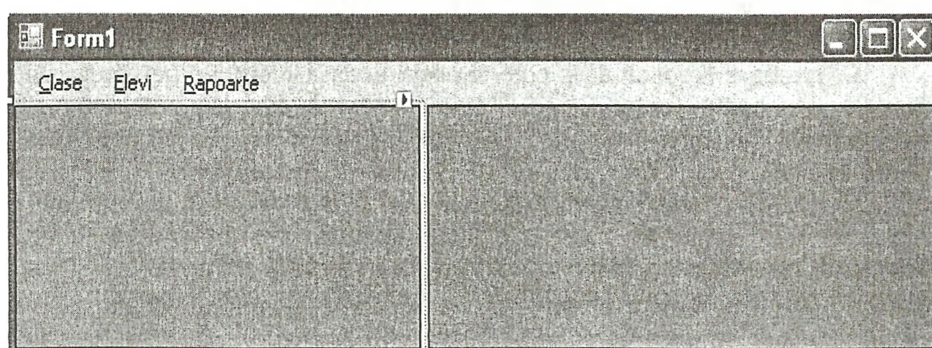
Primary key table:	Foreign key table:
Clase	Elevi
Clasa	Clasa

6. Apăsați OK să închideți dialogul anterior. Următorul dialog vă permite să stabiliți regulile relației de cheie străină. Setați pentru **Delete** și **Update** regula *Cascade*, apoi apăsați OK:

⊟ INSERT And UPDATE Specification	
Delete Rule	Cascade
Update Rule	Cascade
⊟ **Identity**	
(Name)	FK_Elevi_Clase
Description	

7. În *designer*, relația se vizualitează ca o linie cu o cheie spre tabela părinte şi simbolul infinit spre tabela copil (relație *una la mai multe*). Salvați diagrama sub numele *Diagram1*.

8. Plasați pe suprafața formei un control **MenuStrip**, un **SplitContainer**, iar în cele două panouri ale acestuia, câte un control `DataGridView`:



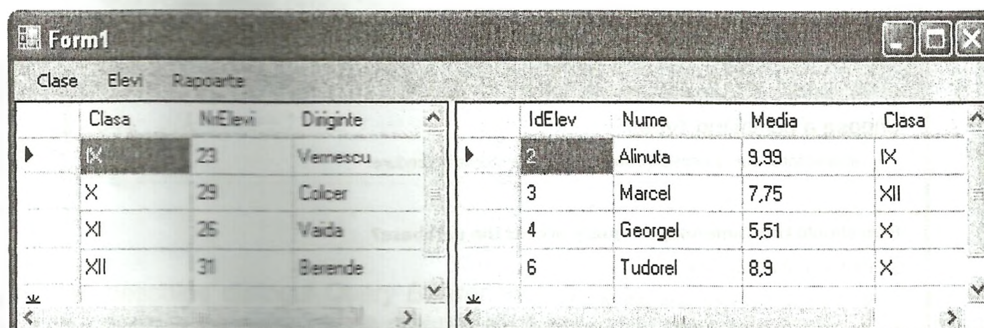
9. Adăugați meniului **Clase**, opțiunile: *Adauga clasa*, *Sterge clasa*, *Modifica clasa*. Adăugați meniului **Elevi**, opțiunile: *Adaugă elev*, *Sterge elev*, *Modifica elev*.
   Adăugați meniului *Rapoarte*, opțiunile: *Diriginti-clase* şi *Media pe clase*.

10. Configurăm *dataset*-ul aplicației cu ajutorul *Wizard*-ului. În fereastra ***Data Sources***, acționați click dreapta pe itemul *ScoalaDataSet* şi selectați *Configure DataSet with Wizard*. În dialogul care se deschide, selectați toate categoriile: tabele, vederi, proceduri stocate, funcții, apoi apăsați *Finish*.

11. Din fereastra ***Data Sources***, trageți cu mouse-ul tabela `Clase` peste gridul din stânga al formei, apoi trageți tabela `Elevi` peste gridul din dreapta.

12. Rulați cu **F5**.

La rulare, se obține:



Interogări. Proceduri stocate.

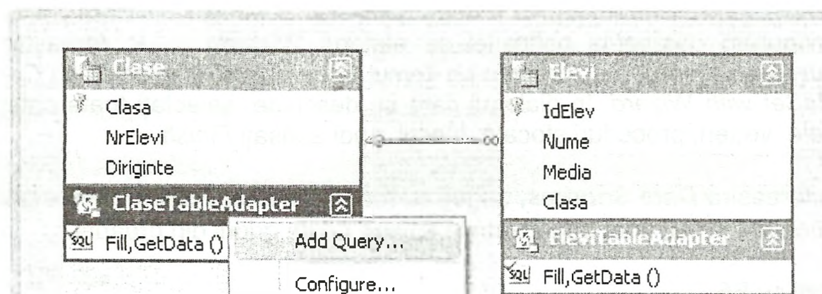
Continuăm să dezvoltăm proiectul **Scoala**, început anterior. Opţiunile din meniuri nu sunt încă funcţionale. Ca să fie, trebuie să interogăm baza de date. Dacă vrem să introducem noi clase sau elevi, avem nevoie de comanda **SQL** `INSERT`. Când vom dori să ştergem din baza de date o clasă sau un elev, aplicăm `DELETE`, iar când modificăm datele ne trebuie `UPDATE`.

O întrebare firească ar fi: cum trimitem aceste interogări serverului de baze de date, din codul nostru C# ? O primă variantă este să manevrăm clasele `SqlConnection`, `SqlCommand`, `SqlDataAdapter`. Este laborios dacă scriem tot codul necesar.

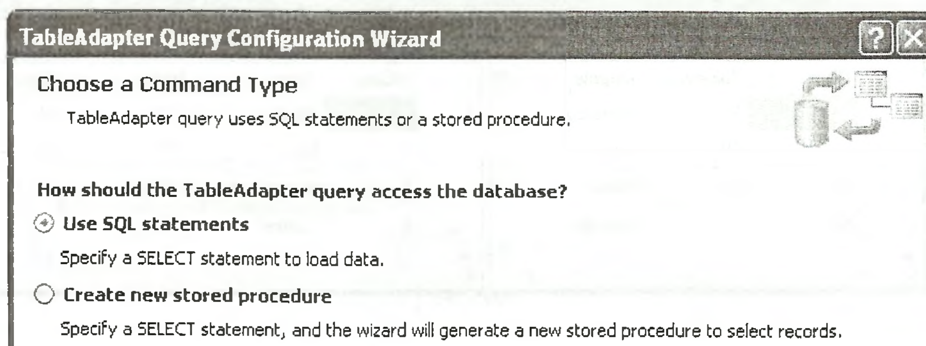
Din fericire, mediul integrat ne ajută mult în această privinţă. Vom genera în mod automat metode C# pentru clasele `ClaseTableAdapter` şi `EleviTableAdapter`. Aceste metode încapsulează interogările SQL sau *proceduri stocate*. În continuare adăugăm interogări aplicaţiei **Scoala**.

Aplicaţia *Scoala* – adăugarea interogărilor

1. În ***Solution Explorer***, acţionaţi dublu click pe itemul `ScoalaDataSet.xsd`. În *Dataset Designer*, pe diagrama tabelei `Clasa`, click dreapta pe `ClaseTableAdapter`. Selectaţi *Add Query...*

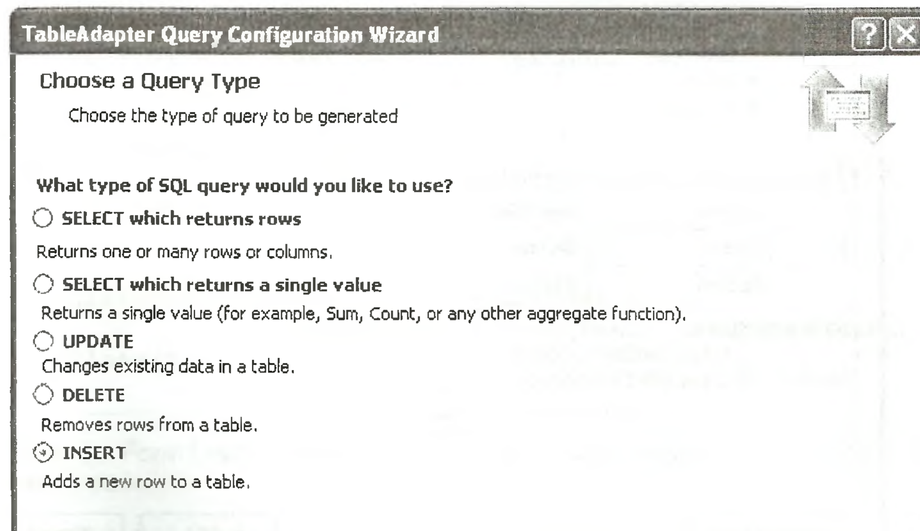


2. Dialogul următor vă permite să generaţi o metodă care conţine o interogare SQL, o procedură stocată, sau o metodă care apelează o procedură stocată existentă:

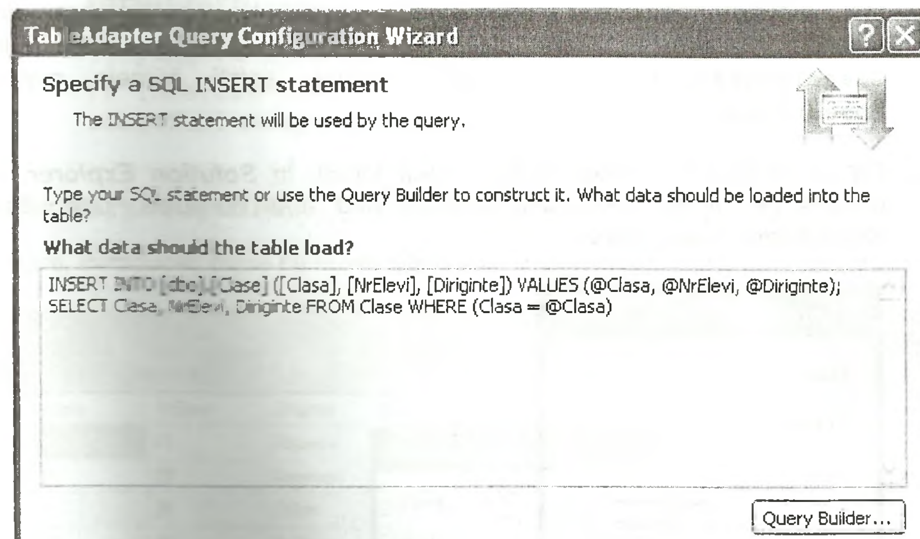


Selectați *Use SQL statements* şi apăsați **OK**.

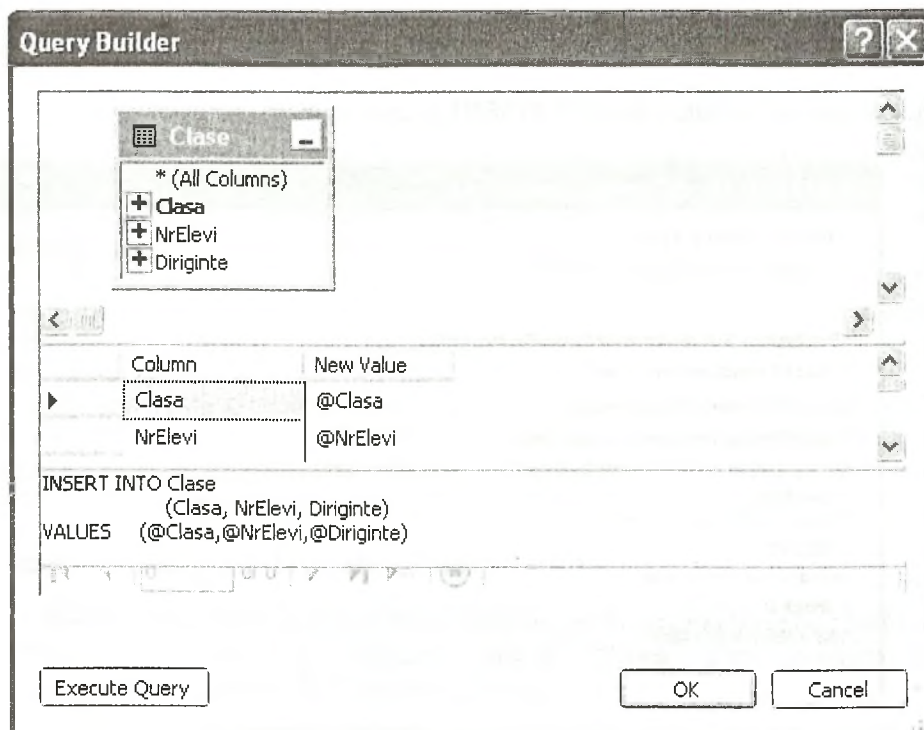
3. În dialogul următor, selectați **INSERT** şi apăsați *Next*:



4. Dialogul care se deschide propune o interogare **INSERT TO** pentru adăugarea unei clase. Apăsați butonul *Query Builder...*

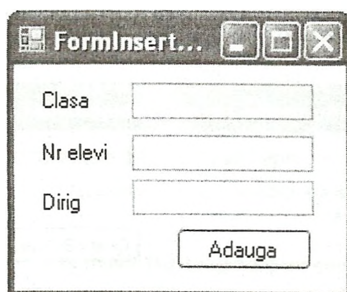


5. Se deschide dialogul *Query Builder*. Acesta este un instrument util, pentru că vă permite să selectați pe diagrama `Clase` câmpurile pe care le inserați, iar codul se generează automat. Evident, puteți edita şi manual:



Ați remarcat modul în care se specifică valorile rândului care se inserează? De fapt, `@Clasa`, `@NrElevi`, `@Diriginte` reprezintă numele **parametrilor metodei** `Insert()` care se va genera, ca membră a clasei `ClaseTableAdapter`. În dialogul următor, setați numele metodei : `InsertClasa`.

6. Ca să introduceți datele, creați o nouă formă. În ***Solution Explorer***, click dreapta pe numele proiectului, selectați *Add*, apoi *Windows Form*... Numiți forma `FormInsertClasa`:



7. Din **Toolbox**, selectați componenta `ClaseTableAdapter` şi plasați-o pe suprafața formei. În *designer tray*, apare referința `claseTableAdapter1`. Facem acest lucru deoarece avem nevoie de adaptor pentru apelul `Insert()`,

dar în clasa **FormInsertClasa** nu este vizibilă adaptorul **claseTableAdapter**, definit în clasa **Form1**.

8. Acționați dublu click pe butonul *Adaugă*. În metoda de tratare, scrieți:

```
private void button1_Click(object sender, EventArgs e)
{
    if (textBox1.Text == "" || textBox2.Text == "" ||
        textBox3.Text == "")
    {
        MessageBox.Show(
                    "Nu ati completat toate campurile!");
        return;
    }

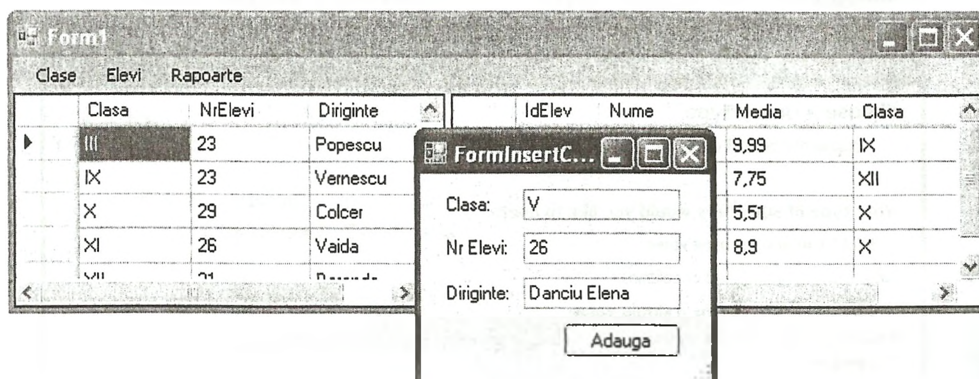
    claseTableAdapter1.Insert(textBox1.Text,
            int.Parse(textBox2.Text), textBox3.Text);
    Close();
}
```

9. Pe forma *Form1*, acționați dublu click pe opțiunea *Adauga clasa*. În metoda de tratare scrieți:

```
private void adaugaClasaToolStripMenuItem_Click(
                              object sender, EventArgs e)
{
     // Instanţiem forma FormInsertClasa
     FormInsertClasa fInsCl = new FormInsertClasa();
     fInsCl.ShowDialog();
     // Acum în baza de date avem un rând nou
     // Încărcăm datele din baza de date în dataset
     claseTableAdapter.Fill(scoalaDataSet.Clase);
}
```

10. Compilați și rulați cu **F5**.

La rulare, introduceți clase cu nume diferite, cu ajutorul opțiunii *Adauga clasa* :



Generarea procedurilor stocate

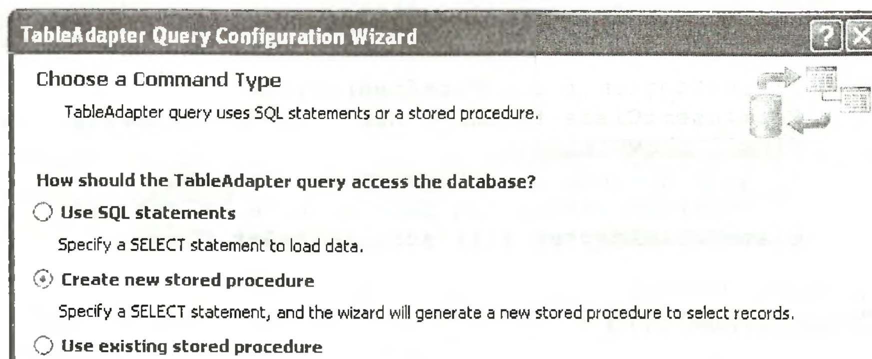
O **procedură stocată** (*stored procedure*) consistă dintr-un grup de comenzi **SQL** adunate sub un nume. Comenzile au fost anterior create şi stocate pe serverul de baze de date. Procedurile stocate sunt compilate o singură dată şi apoi aplicaţia client le poate apela de câte ori este necesar. Sunt foarte rapide. Acceptă date prin parametri de intrare. Datele sunt specificate în timpul execuţiei.

În aplicaţia anterioară, înainte de apelul `Insert()`, am verificat dacă toate câmpurile sunt completate. Nu am verificat alte două aspecte importante: dacă valoarea introdusă pentru *Nr Elevi* este sau nu un număr întreg şi dacă clasa introdusă nu există deja în baza de date. În ambele situaţii, se aruncă excepţii.

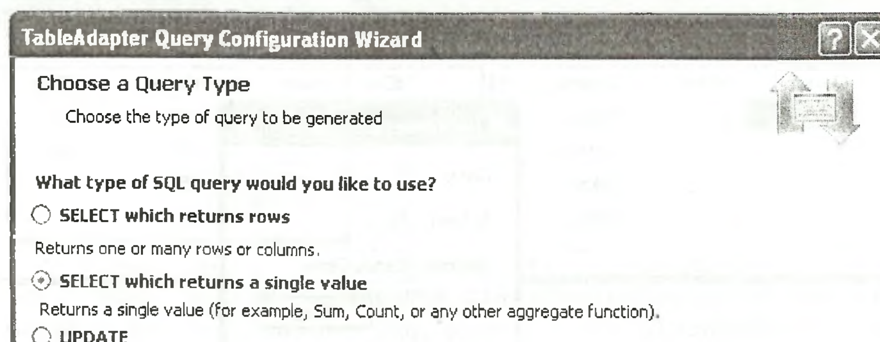
Prima problemă se rezolvă relativ uşor, cu ajutorul metodei `int.TryParse(string, int)`, care încearcă să convertească primul argument în număr, iar dacă nu reuşeşte retureaază *false*.

A doua problemă necesită interogarea bazei de date (un **SELECT**) pentru a vedea dacă clasa nu există deja în baza de date. Pentru aceasta putem genera o nouă metodă în clasa adaptorului, însă vom prefera să creăm o procedură stocată, pentru a vedea cum se procedează. Urmaţi indicaţile :

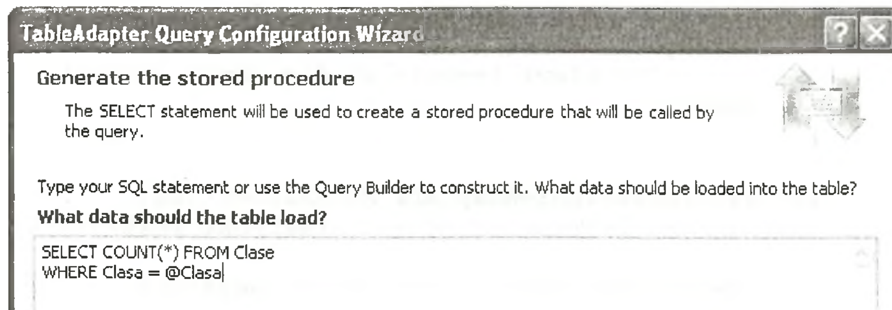
- În *Dataset Designer*, selectaţi diagrama `Clase` şi faceţi click dreapta pe adaptorul `ClaseTableAdapter`. Selectaţi *Add Query*, apoi în dialogul care apare, selectaţi *Create new stored procedure*:



- În dialogul următor, alegeţi "**SELECT wich returns a single value**":



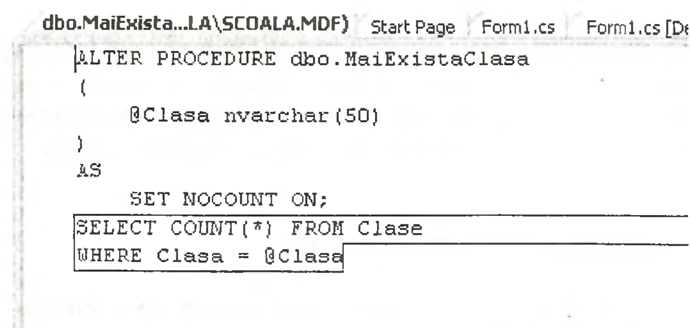
▪ În dialogul următor, editați interogarea astfel:



Am adăugat clauza **WHERE**. Parametrul **@Clasa** devine în mod automat parametru de intrare al procedurii stocate. **COUNT()** returnează numărul de rânduri care conțin clasa **@Clasa**.

▪ Apăsați butonul *Next* şi stabiliți numele procedurii stocate **MaiExistaClasa**. Puteți să păstrați în dialogul următor acelaşi nume şi pentru funcția care apelează procedura stocată. Apăsați butonul *Finish*.

▪ Dacă vreți să vedeți sau să editați codul procedurii stocate, mergeți în ***Database Explorer***, expandați itemul *Stored Procedures* şi dublu click pe procedura **MaiExistăClasa**:



▪ Acum suntem în măsură să completăm codul metodei **button1_Click**:

```
private void button1_Click(object sender, EventArgs e)
{
    if (textBox1.Text == "" || textBox2.Text == "" ||
          textBox3.Text == "")
    {
        MessageBox.Show(
                    "Nu ati completat toate campurile!");
        return;
    }
```

```
        int intreg = 0;
        if (!int.TryParse(textBox2.Text, out intreg))
        {
            MessageBox.Show(
                  "Nr elevi trebuie să fie numar intreg!");
            return;
        }

        if( (int)claseTableAdapter1.MaiExistaClasa(
                                      textBox1.Text) == 1)
        {
             MessageBox.Show("Clasa exista deja!");
             return;
        }
        claseTableAdapter1.Insert(textBox1.Text,
                     int.Parse(textBox2.Text), textBox3.Text);
        Close();
}
```

IMPORTANT !

Metoda `Insert`*, ca dealtfel toate metodele care conțin interogări, pot să arunce excepții. De aceea, trebuie incluse în blocuri* ***try-catch****.*
Noi nu am făcut-o aici, pentru claritate.

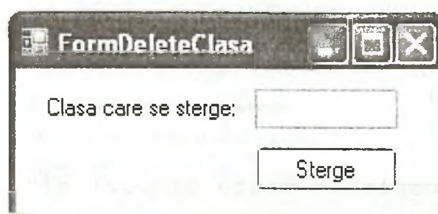
Temă de lucru:

Implementați facilitatea de adăugare a unui nou elev în baza de date la selectarea opțiunii *Adauga elev* din meniul **Elevi**. Trebuie să creeați o funcție suplimentară care interoghează baza de date determinând dacă valoarea pe care o introduce utilizatorul pentru câmpul `Clasa` există sau nu în tabela `Clase`.

Ştergerea unei înregistrări din baza de date

Pentru a şterge o clasă din baza de date, este nevoie de o funcție care încapsulează comanda SQL **DELETE**. Procedăm astfel:

1. Adăugați o nouă formă cu numele `FormDeleteClasa`:



2. Din **Toolbox**, selectați componenta **ClaseTableAdapter** şi plasați-o pe suprafața formei **FormDeleteClasa**. În *designer tray*, apare referința **claseTableAdapter1**.

11. În ***Solution Explorer***, acționați dublu click pe itemul **ScoalaDataSet.xsd**. În *Dataset Designer*, pe diagrama tabelei **Clasa**, click dreapta pe **ClaseTableAdapter**. Selectați *Add Query...*

12. În primul dialog, selectați *Use SQL statements*. Apăsați *Next*. În al doilea dialog, selectați **DELETE**. În *Query Builder*, editați interogarea astfel:

```
DELETE FROM Clase
WHERE      (Clasa = @Original_Clasa)
```

@Original_Clasa este numele parametrului de intrare al funcției care se generează.

13. Stabiliți în următorul dialog, numele funcției: **DeleteClasa**.

14. Pe **Form1**, în *View Designer*, efectuați dublu click pe opțiunea **StergeClasa**. În corpul *handler*-ului introduceți:

```
private void stergeClasaToolStripMenuItem_Click(
                              object sender, EventArgs e)
{
    FormDeleteClasa fDelCl = new FormDeleteClasa();
    fDelCl.ShowDialog();

    // Reîncărcăm cele două tabele din dataset
    // cu datele modificate din baza de date
    claseTableAdapter.Fill(scoalaDataSet.Clase);
    eleviTableAdapter.Fill(scoalaDataSet.Elevi);
}
```

15. Acționați dublu click pe butonul *Sterge*, al formei **FormDeleteElev**. În metoda de tratare introduceți:

```
private void button1_Click(object sender, EventArgs e)
{
    claseTableAdapter1.DeleteClasa(textBox1.Text);
    Close();
}
```

16. Rulați cu **F5**.

La execuție veți constata că atunci când ştergeți o clasă, se şterg şi toate rândurile orfane din tabela **Elevi**, conform regulii de ştergere în cascadă pe care am impus-o pentru menținerea integrității datelor la crearea constrângerii de cheie străină.

Temă de lucru:

Implementaţi facilitatea de ştergere a unui elev, la selectarea opţiunii *Sterge elev* din meniul *Elevi*. Elevul va fi căutat după câmpul **`IdElev`**.

Vederile unei baze de date (Views)

Un ***View*** este o tabelă virtuală, compusă din rezultatul unei interogări. Vederile se crează dinamic. Includ date selectate din una sau mai multe tabele ale bazei de date, şi eventual date rezultate în urma unui calcul.

Modificarea datelor în tabele duce automat la modificarea datelor afişate în vederi. Au avantajul că nu ocupă spaţiu fizic pe disc.

Vederile se folosesc frecvent atunci când se doreşte afişarea rezultatului *join*-ului tabelelor şi când nu se doreşte modificarea datelor ci doar vizualizarea lor. Ne vom lămuri în cele ce urmează.

Aplicaţia *Scoala* – adăugarea vederilor

Vedere cu date selectate dintr-o singură tabelă

Să presupunem că vrem să vedem o tabelă care conţine numai coloanele **`Diriginte`** şi **`Clasa`** din tabela **`Clase`**. Nu este cazul să definim o tabelă nouă. Construim o vedere care afişează doar datele din aceste coloane. Procedaţi astfel:

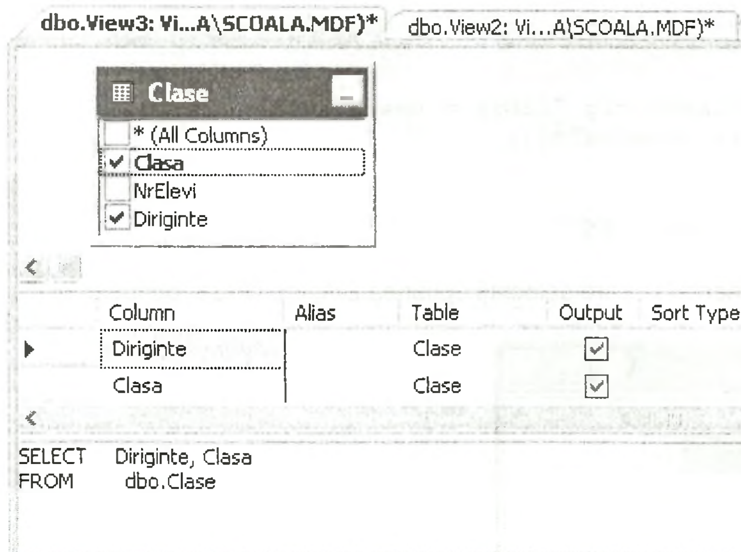
1. Vederea va fi afişată într-un grid. Avem nevoie de o formă nouă. Adăugaţi o formă nouă proiectului, acţionând click dreapta în ***Solution Explorer***, alegeţi *Add* şi selectaţi *Windows Form*... Salvaţi clasa formei cu numele **`FormViewDirigClasa`**.

2. Din **Toolbox** alegeţi un control **`DataGridView`** şi plasaţi-l pe suprafaţa formei.

3. În fereastra ***Database Explorer***, click dreapta pe itemul **Views**, apoi selectaţi *Add New View*.

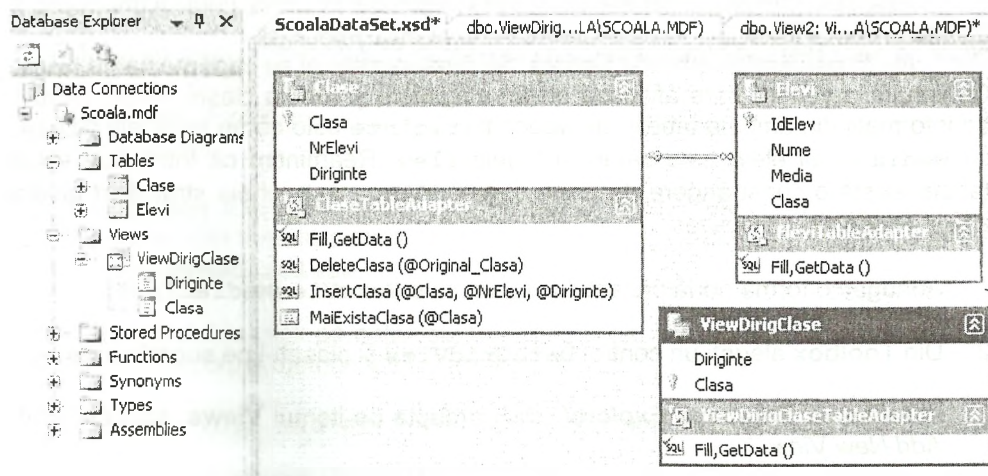
4. În dialogul *Add Table*, selectaţi tabela **`Clase`**, apăsaţi *Add*, apoi click pe butonul *Close*.

5. În *View Designer*, selectaţi câmpul **`Diriginte`** pe diagrama **`Clase`**, apoi câmpul **`Clasa`**. Interogarea **SELECT** se generează în mod automat:



6. Apăsați butonul *Save*, apoi introduceți numele vederii: `ViewDirigClase`.

7. Avem o vedere, dar aceasta nu este încă integrată în *dataset*. Pentru aceasta, deschideți ***DataSet Designer*** cu dublu click pe itemul `ScoalaDataSet.xsd` în ***Solution Explorer***. Din panoul ***Database Explorer***, trageți cu ajutorul mouse-ului vederea pe suprafața designerului. Obțineți:



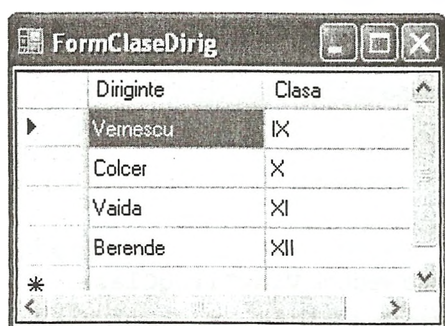
8. În ***Solution Explorer***, dublu click pe forma vederii, pentru a deschide *Form Designer*. Din fereastra ***Data Sources***, trageți vederea pe suprafața formei.

9. Acționați dublu click pe opțiunea *Diriginti-clase* din meniul *Rapoarte*, aflat pe *Form1*. În corpul metodei de tratare scrieți:

```
private void dirigintiClaseToolStripMenuItem_Click(
                               object sender, EventArgs e)
{
      FormClaseDirig fClDir = new FormClaseDirig();
      fClDir.ShowDialog();
}
```

10. Faceți Build și rulați cu **F5**.

La rulare, în momentul în care selectați opțiunea *Diriginti-Clase*, obțineți:

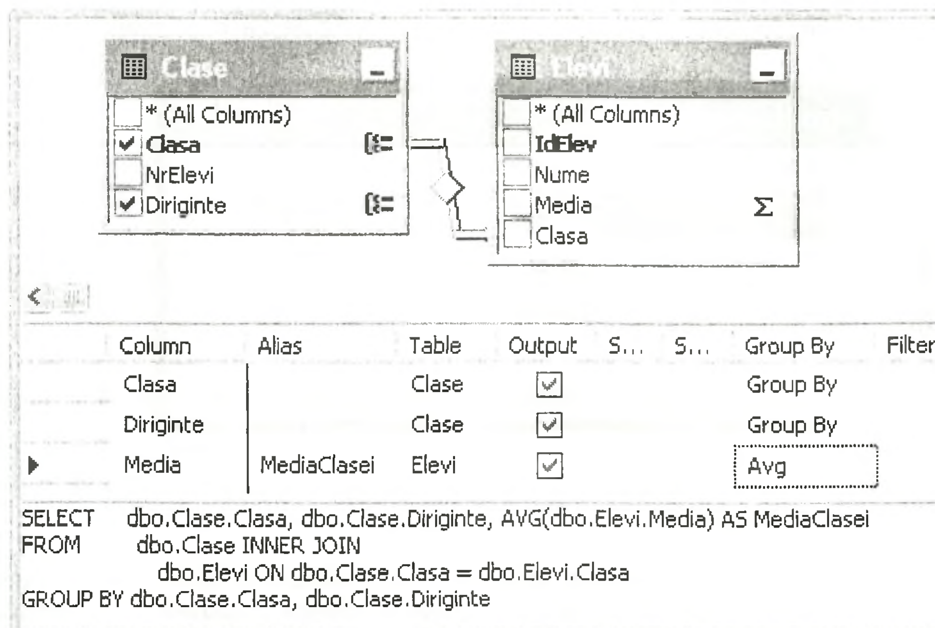


Vedere cu date selectate din două tabele

Aşa cum am spus, tabela temporară rezultată în urma unui **JOIN** poate fi afişată într-un **View**.

Ne întoarcem la aplicația **Şcoala**. Să presupunem ni se cere media pe clase. Construim o vedere care afişează clasa, dirigintele şi media clasei. Avem nevoie de informații din ambele tabele, deoarece **`Diriginte`** este câmp în tabela **`Clase`**, iar **`Media`** (unui elev) este câmp în tabela **`Elev`**. Reamintim că între cele două tabele există o constrângere de cheie străină. **`Clasa`** este cheie străină în tabela **`Elev`**. Procedați ca mai jos:

1. Adăugați o formă nouă proiectului, cu numele **`FormViewMediaClase`**.
2. Din **Toolbox** alegeți un control **`DataGridView`** şi plasați-l pe suprafața formei.
3. În fereastra ***Database Explorer***, click dreapta pe itemul **Views**, apoi selectați *Add New View*.
4. În dialogul *Add Table*, **`Clase`**, adăugați pe rând ambele tabele cu ajutorul butonului *Add*, apoi închideți cu butonul *Close*.
5. În *View Designer*, selectând opțiunile din figura de mai jos, veți obține interogarea necesară:



Evident, puteți să completați și manual codul **SQL**. Observați că se face un *JOIN INTERIOR* care returnează câte un rând pentru fiecare clasă. Media clasei se calculează cu funcția **AVG()**.

6. Apăsați butonul *Save*, apoi introduceți numele vederii: **ViewMediaClase**.

7. Vederea trebuie inserată în *dataset*. Deschideți ***DataSet Designer*** cu dublu click pe itemul **ScoalaDataSet.xsd** în ***Solution Explorer***. Din panoul ***Database Explorer***, trageți cu ajutorul mouse-ului vederea pe suprafața designerului.

11. În ***Solution Explorer***, dublu click pe forma vederii, pentru a deschide *Form Designer*. Din fereastra **Data Sources**, trageți noua vedere pe suprafața formei ***FormViewMediaClase***.

12. Acționați dublu click pe opțiunea *Media pe clase* din meniul *Rapoarte*, aflat pe *Form1*. În corpul metodei de tratare scrieți:

```
private void mediaPeClasaToolStripMenuItem_Click(
                                    object sender, EventArgs e)
{
      // Instanţiem forma şi o afişăm
      FormViewMediaClase fV = new FormViewMediaClase();
      fV.ShowDialog();
}
```

8. Rulați cu **F5**.

La execuție, dacă selectați opțiunea de meniu *Media pe clase*, obțineți:

FormViewMediaClase

Clasa	Diriginte	MediaClasei
IX	Vernescu	9,98999977111
X	Colcer	7,20499992370
XII	Berende	7,75

Probleme propuse

1. Adăugați aplicației **Scoala** un **View** care afişează pe trei coloane: numele diriginților care au elevi cu medii peste o valoare introdusă de la tastatură, numărul de elevi din fiecare clasă care îndeplinesc această restricție şi clasa.

2. Creați o aplicație care întreține o bază de date cu numele ***Firma***. Baza de date conține două tabele: *Clienti* si *Comenzi*. Tabelele vor defini cel putin următoarele coloane: *Clienti*: (*IDClient*, *Nume*, *Prenume*, *Telefon*) si *Comenzi*: (*IDClient*, *Data*, *ValoareComanda*). Aplicația va implementa operații de adăugare de *Clienti* si de *Comenzi*. Va permite de asemenea afişarea tuturor clienților, iar pentru fiecare client, se va afişa valoarea totală a comenzilor sale.

3. Implementați o aplicație care să permită eliminarea dintr-o bază de date a tuturor produselor care au prețul cuprins între două limite introduse de la tastatură. Dacă în baza de date nu există nici o înregistrare cu proprietatea de mai sus, să se afişeze un mesaj. Pentru un produs se memorează: codul, denumirea, prețul, data recepției.

4. Să se realizeze o aplicație care să permită manipularea unei baze de date care conține un tabel cu structura: *nume medicament, compensat sau nu, procent compensare si preț întreg*. Pentru această tabelă să se poate actualiza prețul unui medicament localizat prin nume. Să se realizeze o listă cu medicamentele care nu beneficiază de compensare.

Bibliografie

[1] **Ecma Technical Committee 39 Task Group**
C# Language Specification 4-th Edition, june 2006

[2] **Jesse Liberty**. *Programming C#*. 2-nd Edition. O'Reilly 2002.

[3] **Andrew Troelsen** . *Pro C# 2008 and the .Net 3.5 Platform*. 4-th Edition. Apress 2008.

[4] **Microsoft MSDN Express Library 2008**

[5] **Trey Nash**. *Accelerated C# 2008*. Apress 2008

[6] **Stanley Lippman**. *C# Primer: A Practical Approach*. Pearson Education Inc. 2003

[7] **Herbert Schildt**. *C# 2.0 The Complete Reference*. 2-nd Edition

[8] **Donis Marshall**. *Programming Microsoft Visual C# 2005: The Language*. Microsoft Press 2006.

[9] **Rob Harrop**. *Effective Data Access in C#*. Wrox Press 200

[10] **James Huddleston, Ranga Raghuram**. *Beginning C# 2005 Databases from Novice to Professional*. Apress 2006

[11] **Paul Kimmel.** *Advanced C# Programming*. McGraw Hill/Osborne 2002

[12] **John Sharp, Jon Jagger**. *Microsoft Visual C# .NET Step by Step*. Microsoft Press 2003

[13] **F. Scott Barker**. *Visual C# 2005 Express Edition Starter Kit*. Wiley Publishing Inc. 2005

[14] **Matthew MacDonald**. *Pro .NET 2.0 Windows Forms and Custom Controls*. Apress, 2006.

[15] **John Paul Mueller.** *Visual C# .NET Developer's Handbook*. SYBEX Inc. 2002

[16] **Daniel Solis**. *Illustrated C# 2008*. Apress 2008.

[17] **Matthew MacDonald.** Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second Edition. Apress 2008

[18] **Peter Wright.** *Beginning Visual C# 2005 Express Edition From Novice to Professional*. Apress 2006.

[19] **MAHESH CHAND.** *A Programmer's Guide to ADO.NET in C#*. Apress 2003.

[20] **Mickey Williams.** *Microsoft Visual C# .NET*. Microsoft Corporation 2002